NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AGGREGATE MODELS FOR TARGET ACQUISITION IN URBAN TERRAIN

by

Joseph A. Mlakar

June 2004

Thesis Advisors:	Craig W. Rasmussen Thomas M. Cioppa
Second Reader:	Donovan Phillips

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Aggregate Models for Target Acquisition in Urban Terrain			**5. FUNDING NUMBERS**
6. AUTHOR(S) Joseph A. Mlakar			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Training and Doctrine Analysis Center, Monterey			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

High-resolution combat simulations that model urban combat currently use computationally expensive algorithms to represent urban target acquisition at the entity level. While this may be suitable for small-scale urban combat scenarios, simulation run time can become unacceptably long for larger scenarios. Consequently, there is a need for models that can lend insight into target acquisition in urban terrain for large-scale scenarios in an acceptable length of time.

This research develops urban target acquisition models that can be substituted for existing physics-based or computationally expensive combat simulation algorithms and result in faster simulation run time with an acceptable loss of aggregate simulation accuracy. Specifically, this research explores (1) the adaptability of probability of line of sight estimates to urban terrain; (2) how cumulative distribution functions can be used to model the outcomes when a set of sensors is employed against a set of targets; (3) the uses for Markov Chains and Event Graphs to model the transition of a target among acquisition states; and (4) how a system of differential equations may be used to model the aggregate flow of targets from one acquisition state to another.

14. SUBJECT TERMS Probability of Line of Sight, Line of Sight, Urban Target Acquisition, Cumulative Distribution Functions, Markov Chains, Systems of Differential Equations			**15. NUMBER OF PAGES** 154
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AGGREGATE MODELS FOR TARGET ACQUISITION IN URBAN TERRAIN

Joseph A. Mlakar
Captain, United States Marine Corps
B.S., Carnegie-Mellon University, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

and

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2004

Author: Joseph A. Mlakar

Approved by: Craig W. Rasmussen
Thesis Co-Advisor

Thomas M. Cioppa
Thesis Co-Advisor

Donovan Phillips
Second Reader

James N. Eagle
Chairman, Department of Operations Research

Clyde Scandrett
Chairman, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

High-resolution combat simulations that model urban combat currently use computationally expensive algorithms to represent urban target acquisition at the entity level. While this may be suitable for small-scale urban combat scenarios, simulation run time can become unacceptably long for larger scenarios. Consequently, there is a need for models that can lend insight into target acquisition in urban terrain for large-scale scenarios in an acceptable length of time.

This research develops urban target acquisition models that can be substituted for existing physics-based or computationally expensive combat simulation algorithms and result in faster simulation run time with an acceptable loss of aggregate simulation accuracy. Specifically, this research explores (1) the adaptability of probability of line of sight estimates to urban terrain; (2) how cumulative distribution functions can be used to model the outcomes when a set of sensors is employed against a set of targets; (3) the uses for Markov Chains and Event Graphs to model the transition of a target among acquisition states; and (4) how a system of differential equations may be used to model the aggregate flow of targets from one acquisition state to another.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Due to my large number of "thankees," a bulleted list is probably appropriate.

I would like to thank…

- Lieutenant Colonel Thomas Cioppa for his willingness to allow a Marine to work with TRAC, and for allowing me to become part of the TRAC family during my time at NPS.

- Associate Professor Craig Rasmussen for his advice, direction, and his acceptance to oversee a research project that may not have been his exact area of expertise.

- Major Donovan Phillips for his guidance, contagious work ethic, and mentorship during this research. Never have I learned so much from anyone in such a short period of time. He is an officer worthy of emulation by anyone.

- Professor Gordon Bradley, who taught me everything that I needed to know about Java, thus making the development of the Urban PLOS Prototype Simulation a piece of cake.

- All of the TRAC-WSMR personnel who made it possible to perform LOS experimentation in Combat XXI: Mr. Walt Butler, Mr. Joe Bebbs, Mr. Dave Durda, and Mr. Jason Ladere.

- Mr. Dave McKeown, for his help in mounting urban terrain templates onto an underlying terrain skin.

- My mother, Mary Jo Mlakar, for teaching me how to write in perfect English. This may have been the only case where I wasn't made fun of for having this skill.

- My wife, Jen, for her patience, patience, and patience… and delicious desserts. Jen makes me a better person, whether I like it or not. The work I put in on this thesis doesn't even compare to the work she does at home; I am forever in debt to her sacrifices for our family.

But most of all, I must thank my son, Payton, for his remarkable ability to keep my spirits up in times of difficulty. Whenever I needed a push in the right direction, I could always look to him for inspiration. Payton, this thesis is written for you; may you always acquire every target you wish.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Emerging Army organization and operational concepts are leading the movement toward a lighter, more agile force – a force that is increasingly dependent upon multi-sensor collection of information across the battlespace. As a result, today's Army has become more reliant upon information dominance in order to achieve its objectives. A large component of information dominance on the modern battlefield lies in the field of urban target acquisition. The outcomes of current and future military operations will be highly correlated with urban target acquisition capabilities. Thus, in order to model modern combat effectively, current combat simulations must accurately represent target acquisition in urban terrain. While urban target acquisition may be a simple concept to understand, it is certainly a complicated concept to model.

Currently, high-resolution combat simulations that model urban combat use computationally expensive algorithms to model urban target acquisition at the entity level. While this may be suitable for small-scale urban combat scenarios, simulation run time can become unacceptably long for large-scale urban combat scenarios. Consequently, there is a need for models that can lend insight into target acquisition in urban terrain for large-scale scenarios in an acceptable length of time. Such models could be used to improve simulation run time or be employed where aggregate target acquisition models were previously non-existent.

The objective of this research is to develop urban target acquisition models that can be substituted for existing physics-based or computationally expensive combat simulation algorithms and result in faster simulation run time with an acceptable loss of aggregate simulation accuracy. Our approach to accomplishing this objective is to apply mathematical modeling tools in novel ways to represent urban target acquisition. Our focus is not to develop these models to completion; rather, it is our intent to demonstrate the utility of our approach and methodology in order to direct future research efforts.

Our research concentrates on two distinct areas: (1) adapting probability of line of sight estimates to urban terrain and (2) developing stochastic and analytical models that can lend insight into urban target acquisition.

First, we investigate the adaptability of probability of line of sight estimates to urban terrain. This is done through the development of a prototype simulation that estimates probabilities of line of sight in urban terrain for a given urban terrain type, sensor elevation, and sensor-to-target range. Furthermore, we demonstrate the usefulness of this simulation through the generation of urban probability of line of sight curves. Additionally, we present some emerging topics unique to urban probability of line of sight to which further research should be dedicated.

Second, we examine stochastic and analytical models that may lend insight into target acquisition in urban terrain. Specifically, we explore how (1) cumulative distribution functions can be used to model the outcomes when a set of sensors is employed against a set of targets; (2) Markov Chains and Event Graphs can be used to model the transition of a target among acquisition states; and (3) a system of differential equations may be used to model the aggregate flow of targets from one state to another. Our intent is to determine which of these proposed models shows promise in developing faster simulation algorithms. For those models that do show promise, we develop them further and explain their utility in a combat simulation that models large-scale urban combat scenarios.

I. INTRODUCTION

A. OVERVIEW

Emerging Army organization and operational concepts are leading the movement toward a lighter, more agile force – a force that is increasingly dependent upon multi-sensor collection of information across the battlespace. As a result, today's Army has become more reliant upon information dominance in order to achieve its objectives. A large component of information dominance on the modern battlefield lies in the field of urban target acquisition. The outcomes of current and future military operations will be highly correlated with urban target acquisition capabilities. Thus, in order to model modern combat effectively, current combat simulations must accurately represent target acquisition in urban terrain. While urban target acquisition may be a simple concept to understand, it is certainly a complicated concept to model.

Currently, high-resolution combat simulations that model urban combat use computationally expensive algorithms to model urban target acquisition at the entity level. While this may be suitable for small-scale urban combat scenarios, simulation run time can become unacceptably long for large-scale urban combat scenarios. Consequently, there is a need for models that can lend insight into target acquisition in urban terrain for large-scale scenarios in an acceptable length of time. Such models could be used to improve simulation run time or be employed where aggregate target acquisition models were previously non-existent.

B. THESIS OBJECTIVES

The objective of this research is to develop urban target acquisition models that can be substituted for existing physics-based or computationally expensive combat simulation algorithms and result in faster simulation run time with an acceptable loss of aggregate simulation accuracy. Our approach to accomplishing this objective is to apply mathematical modeling tools in novel ways to represent urban target acquisition. Our focus is not to develop these models to completion; rather, it is our intent to demonstrate the utility of our approach and methodology in order to direct future research efforts.

C. THESIS SCOPE

Our research concentrates on two distinct areas: (1) adapting probability of line of sight estimates to urban terrain and (2) developing stochastic and analytical models that can lend insight into urban target acquisition.

First, we investigate the adaptability of probability of line of sight estimates to urban terrain. This is done through the development of a prototype simulation that estimates probabilities of line of sight in urban terrain for a given urban terrain type, sensor elevation, and sensor-to-target range. Furthermore, we demonstrate the usefulness of this simulation through the generation of urban probability of line of sight curves. Additionally, we present some emerging topics unique to urban probability of line of sight to which further research should be dedicated.

Second, we examine stochastic and analytical models that may lend insight into target acquisition in urban terrain. Specifically, we explore how (1) cumulative distribution functions can be used to model the outcomes when a set of sensors is employed against a set of targets; (2) Markov Chains and Event Graphs can be used to model the transition of a target among acquisition states; and (3) a system of differential equations may be used to model the aggregate flow of targets from one state to another. Our intent is to determine which of these proposed models shows promise in developing faster simulation algorithms. For those models that do show promise, we develop them further and explain their utility in a combat simulation that models large-scale urban combat scenarios.

D. THESIS SPONSORS

This research is funded by the Army Model and Simulation Office (AMSO) and is further sponsored by the Urban Operations Focus Area Collaborative Team (UO FACT) and the Army's Training and Doctrine Command Analysis Center, Monterey (TRAC-MTRY).

II. PROBABILITY OF LINE OF SIGHT IN URBAN TERRAIN

A. BACKGROUND

In order for a combat simulation to model modern combat accurately, it must incorporate methods to evaluate whether or not one entity detects another entity on the battlefield. While detection is certainly a simple concept to understand, it is not always a simple concept to model. Detection is dependent upon numerous factors; one of these factors is line of sight (LOS).

The LOS factor is simply a yes or no answer: does one entity have LOS to another entity? Various algorithms exist for computing LOS in open terrain; the most widely used open terrain LOS algorithms are discussed in (Champion, 1995) and (Champion, 2002). While these algorithms have subtle differences, almost all open terrain LOS algorithms use a common basis to determine whether a sensor has LOS to a target: comparison of slopes. First, using given elevation data, the slope of the line connecting the sensor to the target is computed. Next, for sufficiently many intermediate points on the ground directly between the sensor and the target, the slope of the line connecting the sensor to the intermediate point is computed (this process of computing "intermediate" slopes is usually accomplished by successively "stepping" along the underlying terrain skin from the sensor location to the target location). If any of these intermediate slopes exceeds the slope from the sensor to the target, then the sensor does not have LOS to the target. Figure 1 shows a simplified illustration of LOS determination in open terrain.

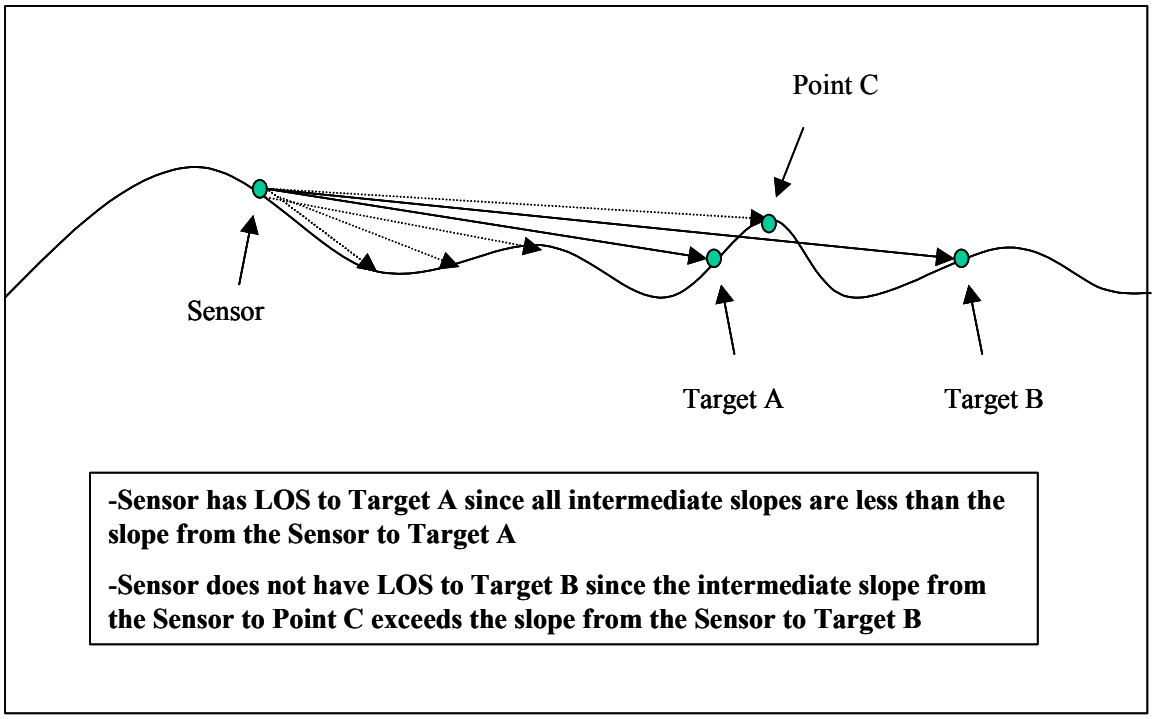


Figure 1. Illustration of LOS Determination in Open Terrain

Computing LOS in urban terrain is even more complicated, as man-made structures, such as buildings, can block LOS. Additionally, targets might be located inside man-made structures, effectively blocking LOS from any location. Thus, an algorithm to compute LOS in urban terrain must evaluate whether LOS is blocked by the underlying terrain skin *and*, at a minimum, man-made structures (more sophisticated algorithms would consider vegetation, obscurants, etc.). Figure 2 demonstrates the complexities introduced to LOS by man-made structures in urban terrain.

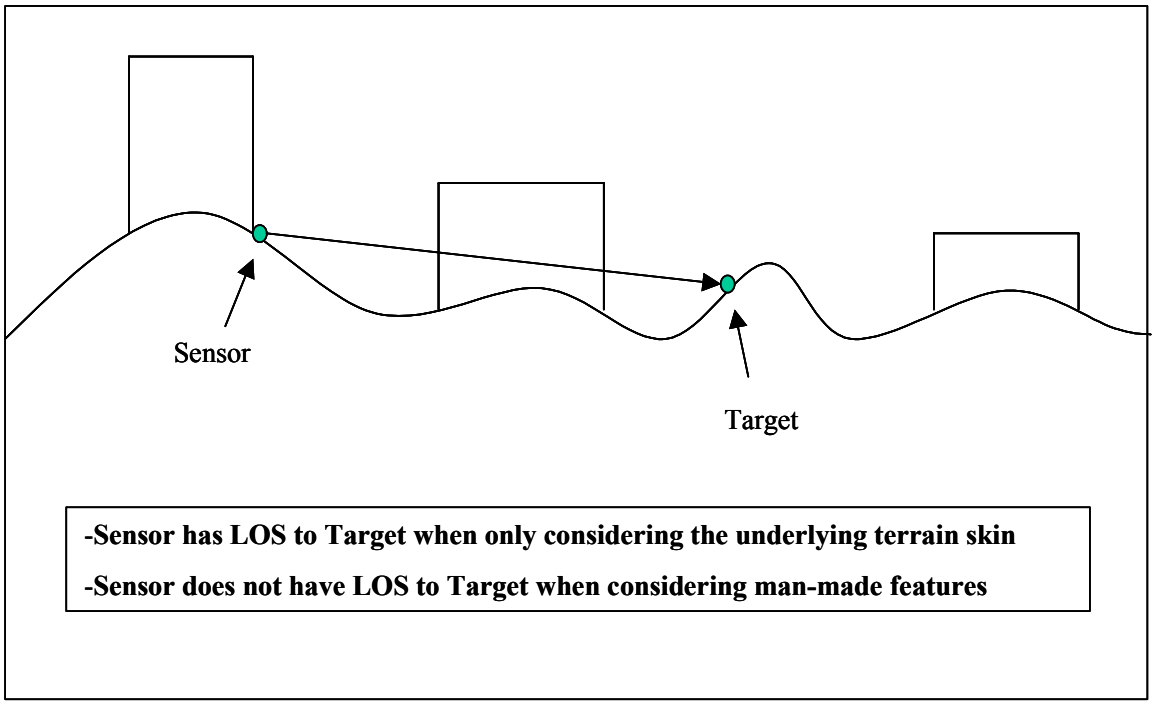


Figure 2. Complexities Introduced to LOS by Urban Terrain

1. Probability of Line of Sight

As one can imagine, LOS algorithms can become quite complex in combat simulations. Combine this with the fact that, by nature, LOS calculations must be frequently executed, and one can understand why LOS algorithms can consume a large amount of computing time in a combat simulation. As a result, researchers have focused a large amount of effort to find more efficient or equally useful methods for computing LOS. One of these approaches is to use probability of line of sight (PLOS) in place of traditional LOS calculations.

PLOS calculations use digital elevation data to estimate the probability that LOS exists at a given sensor-to-target range (this is *ground* range, not slant range; see Figure 3), sensor elevation, and terrain type. A combat simulation using a PLOS methodology would calculate PLOS estimates for each sensor-to-target range, sensor elevation, and terrain type in pre-processing (prior to the actual simulation run) and store them in a PLOS "look-up table." Then, when the simulation needs to make an LOS determination

during the simulation run, it simply uses the given sensor-to-target range, sensor elevation, and terrain type to determine the probability that LOS exists using the PLOS look-up table. Subsequently, a simple uniform random number draw and comparison to the PLOS will determine whether LOS exists.

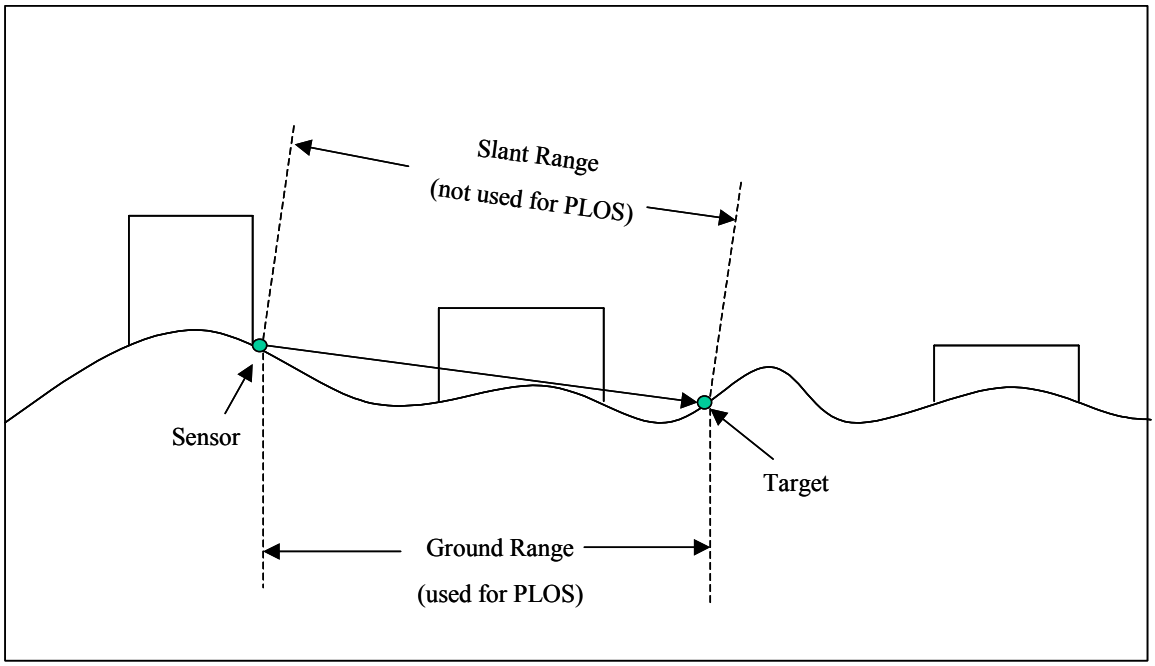


Figure 3. Illustration of Sensor-To-Target Range

Thus, PLOS requires the same data as traditional LOS algorithms, but it is less accurate (a traditional LOS algorithm is deterministic, while computing LOS using PLOS estimates is stochastic). The reason, then, for even considering the use of PLOS is that traditional LOS algorithms necessitate the use of computationally expensive computations *during the simulation*, increasing run time. PLOS calculations, in contrast, require the use of computationally expensive calculations *during pre-processing* and computationally inexpensive table look-ups during the simulation. Thus, if there is an ample pre-processing time budget for a simulation, the use of a PLOS methodology can greatly reduce simulation run time.

PLOS is clearly not appropriate for simulations involving few entities and few events or interactions – traditional LOS algorithms will suffice. However, when the number of entities and interactions becomes large, simulation run time can become unacceptably long. In these situations, it may be worthwhile to sacrifice LOS accuracy in order to reduce simulation run time. It is in exactly these situations that a PLOS methodology can be useful.

2. PLOS Curves

A PLOS curve is simply a plot of PLOS versus sensor-to-target range. All other variables, such as sensor elevation, target elevation, and terrain type, are fixed for a given PLOS curve. Thus, PLOS curves consolidate PLOS estimates in graphical form and provide a visual picture of how PLOS behaves as a function of range. An example of a PLOS curve is shown in Figure 4. PLOS curves will, in general, be monotonic decreasing – we would expect PLOS to decrease as range increases. The shape of PLOS curves, however, will vary. PLOS curves may be linear, piecewise linear, concave, convex, or a combination of these shapes (as shown in Figure 4).

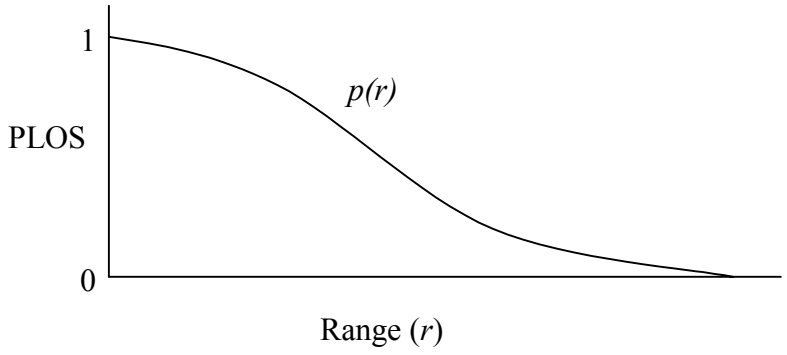


Figure 4. Example of a PLOS Curve

3. Joint Warfare System Use of PLOS

The Joint Warfare System (JWARS) is a campaign-level model of military operations that has been developed for use by the Office of the Secretary of Defense, the Joint Staff, the Services, and the Warfighting Commands (Stone, 2001). Because of a willingness to sacrifice LOS accuracy in order to improve simulation run time, JWARS currently employs a PLOS methodology for combat scenarios in open terrain. In JWARS, PLOS is a function of three parameters: sensor-to-target range, sensor elevation, and effective terrain roughness. The third parameter, effective terrain roughness, is a function of sensor-to-target range and the aggregate qualities of the underlying terrain; more detailed information on the calculation of effective terrain roughness can be found in (Blacksten, 2002) and (Blacksten, 2004).

The PLOS estimates computed by JWARS are approximated by fitting a functional approximation to the PLOS data (using the sensor-to-target range, sensor elevation, and effective terrain roughness as parameters) rather than using a PLOS look-up table. Consequently, when JWARS needs to perform a LOS calculation, it does not explicitly calculate LOS based on a comparison of slopes. Instead, it simply evaluates a function in order to determine the probability that LOS exists – the PLOS. Then, as previously described, a uniform random number draw and comparison to the PLOS will stochastically determine whether LOS exists.

As demonstrated by JWARS, the use of PLOS in combat simulations that model open terrain is not a novel idea; however, at the time of this research, a PLOS methodology had yet to be applied to urban terrain.

B. OBJECTIVES

The ultimate objective of this research is to determine the usefulness of a PLOS methodology for urban combat simulations. In order to accomplish this, our intent is to develop a product that allows for rapid LOS determination for many sensors and many targets in an urban combat simulation through the generation and use of urban PLOS look-up tables or closed-form functions.

Specifically, our research focuses on the following goals:

1. Explore how the PLOS methodology developed for open terrain can be adapted to produce PLOS estimates for urban terrain.

2. Develop a prototype simulation that will generate PLOS estimates for a given range, sensor elevation, and urban terrain type.

3. Determine how urban PLOS behaves as a function of range, sensor elevation, and urban terrain type via analysis of urban PLOS curves.

4. Identify topics unique to urban PLOS to which further research should be dedicated.

C. THE URBAN PLOS PROTOTYPE SIMULATION

1. Urban PLOS Parameters

In leveraging the PLOS methodology employed by JWARS for open terrain, we continue to use range and sensor elevation as the two baseline parameters for computing urban PLOS estimates. However, when considering urban terrain, where buildings would seem to be the predominant obstruction to LOS, terrain roughness does not seem to be an appropriate parameter. In place of the terrain roughness factor, we use an Urban Terrain Zone (UTZ) classification as the third factor in our computations of urban PLOS estimates.

2. Urban Terrain Classification

Richard Ellefsen of San Jose State University developed the UTZ classification system currently used by the United States Army. UTZ classifications are determined by factors such as building dimensions, building spacing, construction types, and street widths. Additional information on UTZs and the UTZ classification system can be found in (Ellefsen, 1987). The UTZ classifications suggested by Ellefsen essentially provide a set of "rules" which a particular urban area is expected to obey in order to earn a given UTZ classification. Thus, many different urban areas may all be assigned the same UTZ classification, as long as they all conform to the rule set proposed for the given UTZ classification.

Army Materiel Systems Analysis Activity (AMSAA) and TerraSim have been working to construct *geotypical urban templates*: terrain templates that obey the rule sets proposed by Ellefsen, and, thus, are representative of each UTZ. These geotypical urban templates are not representative of any specific urban area; they only characterize a particular UTZ classification. Additionally, these geotypical urban templates do not have an underlying terrain skin – they simply display buildings resting on a flat surface. It is exactly these geotypical urban templates that we used to perform our urban PLOS experiments.

3. Simulation Assumptions and Limitations

Before our development of the prototype simulation, we make several simplifying assumptions. Once the simulation is coded and the results validated, these assumptions can then be relaxed in order to provide additional insight into urban PLOS.

1. Only buildings can block LOS (the effect of underlying terrain skin, vegetation, vehicles, lampposts, etc. is omitted).

2. Buildings can be of any shape, but each building's height must be uniform (each building must have a flat roof).

3. LOS is considered in all directions (360 degrees) from each sensor location.

4. Sensors and targets are not located inside buildings – only LOS exterior to buildings is considered.

5. Only ground-level targets are considered.

4. Data

At the time of this research, there were three geotypical urban templates available on which we could perform urban PLOS experiments. These geotypical urban templates were each 750 meters long by 750 meters wide and were named the City Core, Outlying High-Rise Area, and Commercial Ribbon. Overhead views of these three urban templates are shown in Figures 5, 6, and 7, respectively. The City Core urban template is densely populated with high-rise buildings. The Outlying High-Rise Area contains

buildings of similar heights to those of the City Core, but is more sparsely populated with buildings. The Commercial Ribbon is sparsely populated with smaller buildings (no more than six stories tall) and contains a central main street area. The minimum building height, maximum building height, mean building height, and fraction of land area covered by buildings in each urban template is summarized in Table 1.

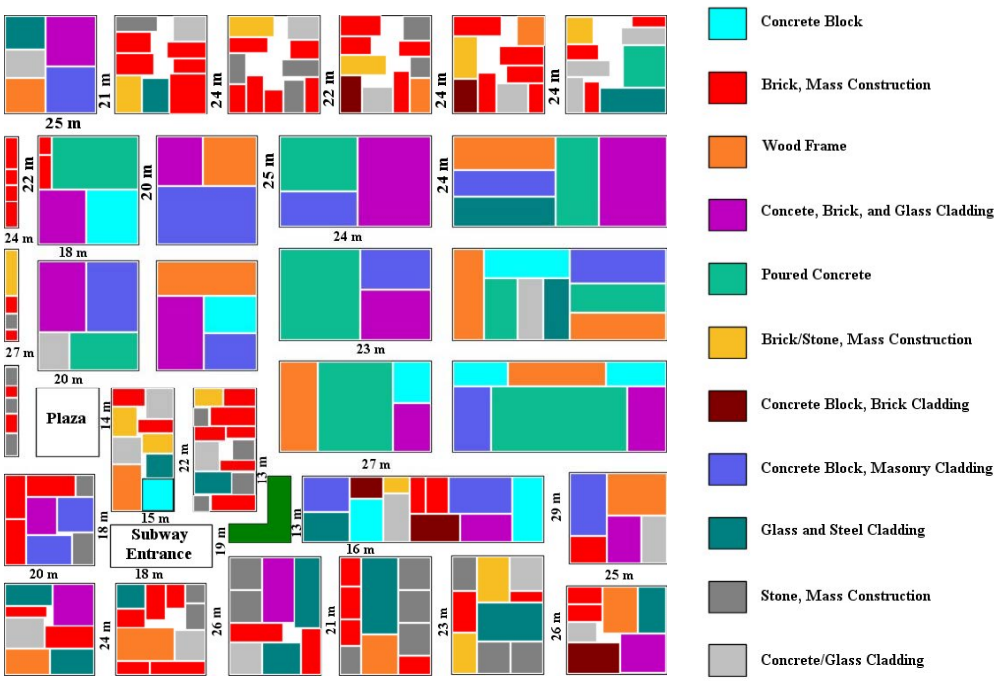


Figure 5. Overhead View of City Core Urban Template (From ref. Fordyce, 2003)

Figure 6. Overhead View of Outlying High-Rise Area Urban Template (From ref. Fordyce, 2003)

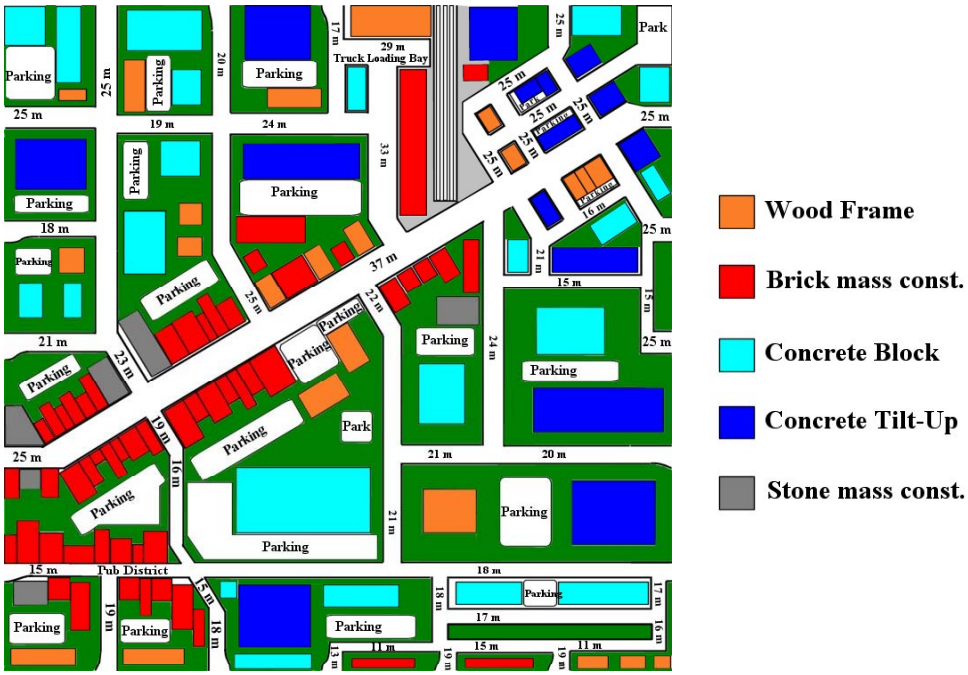


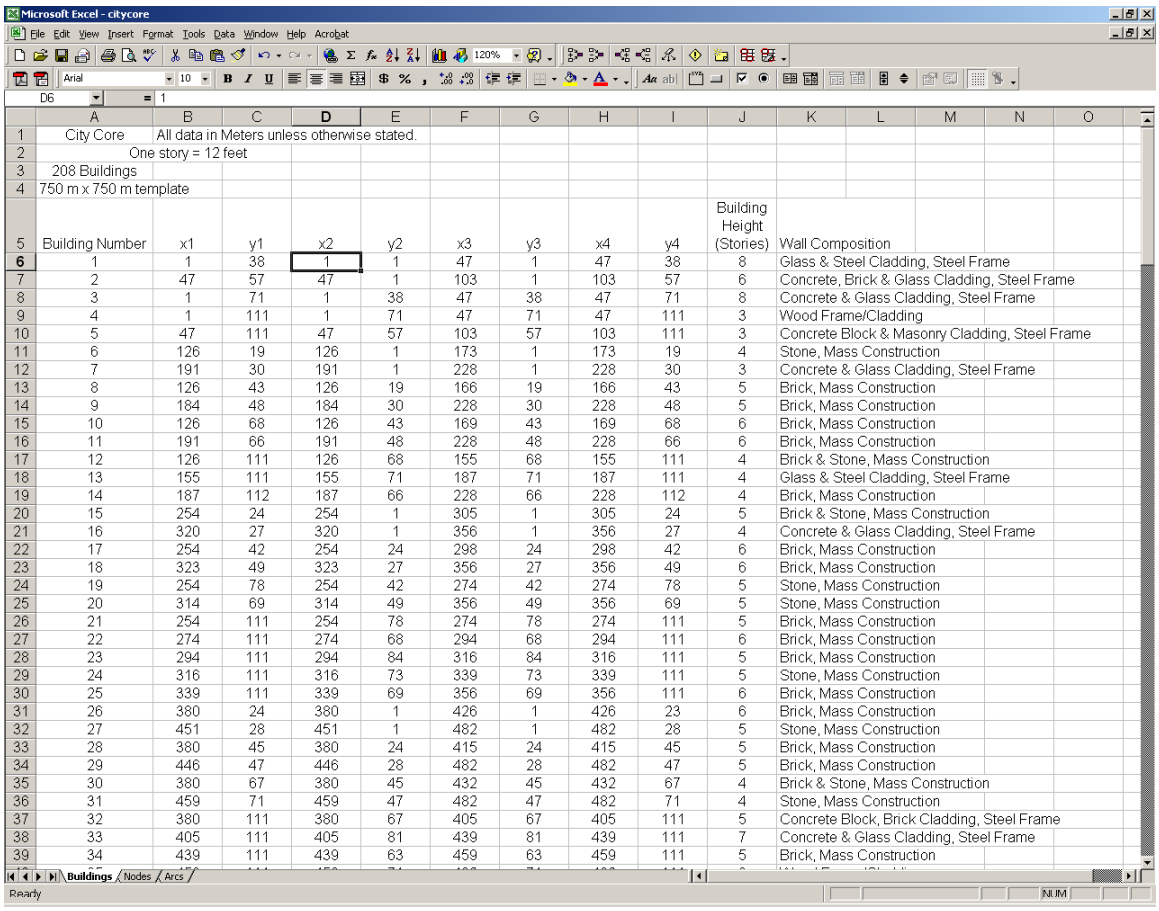
Figure 7. Overhead View of Commercial Ribbon Urban Template (From ref. Fordyce, 2003)

	Min Building Height (meters)	Max Building Height (meters)	Mean Building Height (meters)	Coverage Factor*
City Core	10.97	146.30	31.13	0.57
Outlying High-Rise Area	14.63	109.73	39.13	0.26
Commercial Ribbon	3.66	21.95	9.82	0.30

***Fraction of Ground Covered By Buildings**

Table 1. Summary Statistics for Each Urban Template

The data used for our urban PLOS experiments is in the form of a Microsoft Excel spreadsheet with building locations and dimensions for each geotypical urban template. An example of this data for the City Core urban template is shown in Figure 8. Each line of the spreadsheet contains information on one particular building – it details (1) the x- and y-coordinate of each of the four corners of the building, and (2) the height of the building in stories. The wall composition of each building is also included in these files, but it is not used for our calculations of urban PLOS estimates. These spreadsheets are converted into standard text files and used as the input files for our prototype simulation.



	A	B	C	D	E	F	G	H	I	J	K
1	City Core	All data in Meters unless otherwise stated.									
2		One story = 12 feet									
3	208 Buildings										
4	750 m x 750 m template										
5	Building Number	x1	y1	x2	y2	x3	y3	x4	y4	Building Height (Stories)	Wall Composition
6	1	1	38	1	1	47	1	47	38	8	Glass & Steel Cladding, Steel Frame
7	2	47	57	47	1	103	1	103	57	6	Concrete, Brick & Glass Cladding, Steel Frame
8	3	1	71	1	38	47	38	47	71	8	Concrete & Glass Cladding, Steel Frame
9	4	1	111	1	71	47	71	47	111	3	Wood Frame/Cladding
10	5	47	111	47	57	103	57	103	111	3	Concrete Block & Masonry Cladding, Steel Frame
11	6	126	19	126	1	173	1	173	19	4	Stone, Mass Construction
12	7	191	30	191	1	228	1	228	30	3	Concrete & Glass Cladding, Steel Frame
13	8	126	43	126	19	166	19	166	43	5	Brick, Mass Construction
14	9	184	48	184	30	228	30	228	48	5	Brick, Mass Construction
15	10	126	68	126	43	169	43	169	68	6	Brick, Mass Construction
16	11	191	66	191	48	228	48	228	66	6	Brick, Mass Construction
17	12	126	111	126	68	155	68	155	111	4	Brick & Stone, Mass Construction
18	13	155	111	155	71	187	71	187	111	4	Glass & Steel Cladding, Steel Frame
19	14	187	112	187	66	228	66	228	112	4	Brick, Mass Construction
20	15	254	24	254	1	305	1	305	24	5	Brick & Stone, Mass Construction
21	16	320	27	320	1	356	1	356	27	4	Concrete & Glass Cladding, Steel Frame
22	17	254	42	254	24	298	24	298	42	6	Brick, Mass Construction
23	18	323	49	323	27	356	27	356	49	6	Brick, Mass Construction
24	19	254	78	254	42	274	42	274	78	5	Stone, Mass Construction
25	20	314	69	314	49	356	49	356	69	5	Stone, Mass Construction
26	21	254	111	254	78	274	78	274	111	5	Brick, Mass Construction
27	22	274	111	274	68	294	68	294	111	6	Brick, Mass Construction
28	23	294	111	294	84	316	84	316	111	5	Brick, Mass Construction
29	24	316	111	316	73	339	73	339	111	5	Stone, Mass Construction
30	25	339	111	339	69	356	69	356	111	6	Brick, Mass Construction
31	26	380	24	380	1	426	1	426	23	6	Brick, Mass Construction
32	27	451	28	451	1	482	1	482	28	5	Stone, Mass Construction
33	28	380	45	380	24	415	24	415	45	5	Brick, Mass Construction
34	29	446	47	446	28	482	28	482	47	5	Brick, Mass Construction
35	30	380	67	380	45	432	45	432	67	4	Brick & Stone, Mass Construction
36	31	459	71	459	47	482	47	482	71	4	Stone, Mass Construction
37	32	380	111	380	67	405	67	405	111	5	Concrete Block, Brick Cladding, Steel Frame
38	33	405	111	405	81	439	81	439	111	7	Concrete & Glass Cladding, Steel Frame
39	34	439	111	439	63	459	63	459	111	5	Brick, Mass Construction

Figure 8. Microsoft Excel Spreadsheet with Building Data for City Core Urban Template

5. Simulation Implementation

The Urban PLOS Prototype Simulation (UPPS) that we developed to compute PLOS estimates for urban terrain is written in the Java programming language using an object-oriented approach. The UPPS Java code only generates PLOS estimates; it does not construct PLOS curves. The statistical package S-Plus was used to generate the PLOS curves depicted in this thesis; for more information on S-Plus see (Notes on S-Plus, 2004). Appendix A contains a "User's Manual" with more detailed information on the development and the instructions for use of the UPPS. The Java code of the UPPS can be found in Appendix B.

The UPPS employs a Monte Carlo simulation in order to determine PLOS estimates for each range, sensor elevation, and terrain type. First, building data for an

urban template (similar to that in Figure 8) is provided to the UPPS as an input file. Next, the sensor elevation is fixed. One sensor (at the fixed elevation) and one target (at zero elevation) are uniformly randomly placed in the urban template. Then, a traditional LOS algorithm, using comparison of slopes, is applied to determine if the sensor has LOS to the target. This process is then replicated for a sufficiently large, user-determined number of sensor-target pairs. The results are sorted by the range between sensor and target and placed in range bins. Subsequently, the fraction of sensor-target pairs for which LOS is achieved in each range bin is calculated; this quantity is the PLOS for the given range bin, sensor elevation, and terrain type. An illustration of how these PLOS values are calculated is shown in Figure 9. This entire process is then repeated for a new, fixed sensor elevation.

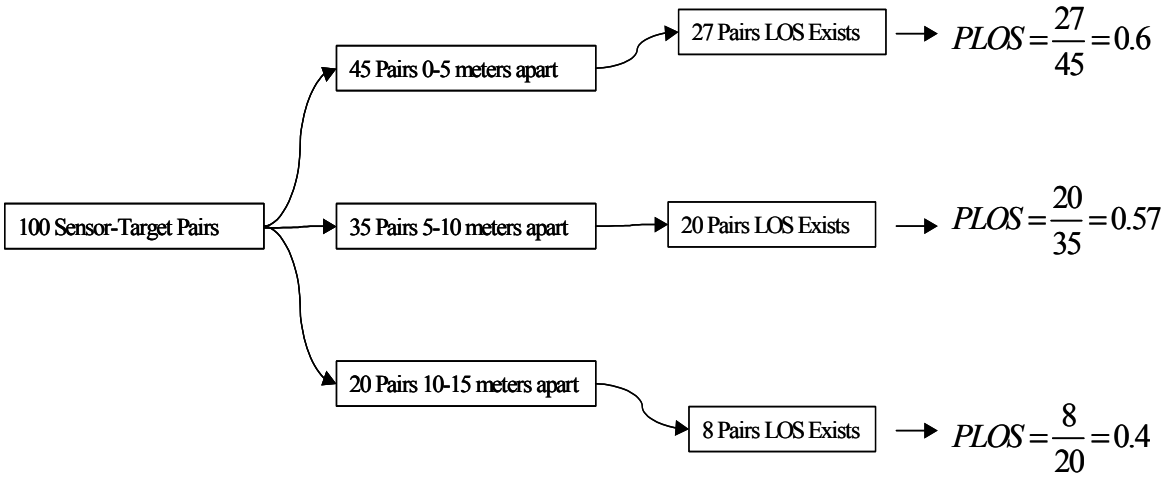


Figure 9. Illustration of PLOS Calculations in the UPPS

D. RESULTS AND ANALYSIS

1. PLOS Curves for each Urban Template

For each of the three geotypical urban templates available, we used the UPPS to calculate PLOS estimates for several different sensor elevations and ranges. For each sensor elevation, ten million different sensor-target pairs were used to calculate PLOS estimates. These ten million sensor-target pairs were sorted into bins of five-meter size (0 – 5 meter bin, 5 – 10 meter bin, etc.). Figure 10 is a plot showing, for one particular sensor elevation and terrain type combination, the number of sensor-target pairs resulting in each range bin; a plot for all other sensor elevation/terrain type combinations exhibited the same shape. When analyzing the results, ranges longer than 750 meters were not considered because of the limiting size of the urban templates – the templates were not large enough to provide a sufficient variety in the geometry of line segments longer than 750 meters. While the shape of the histogram in Figure 10 is not "uniform," the large number of sensor-target pairs generated virtually ensured that each bin of line segments shorter than 750 meters would contain at least 1000 observations – a sufficient number of observations for the purposes of this work. Follow-on work in this area could focus on refining the UPPS to generate a more uniform distribution of ranges. The output data from the UPPS was then used to generate PLOS curves for each terrain type.

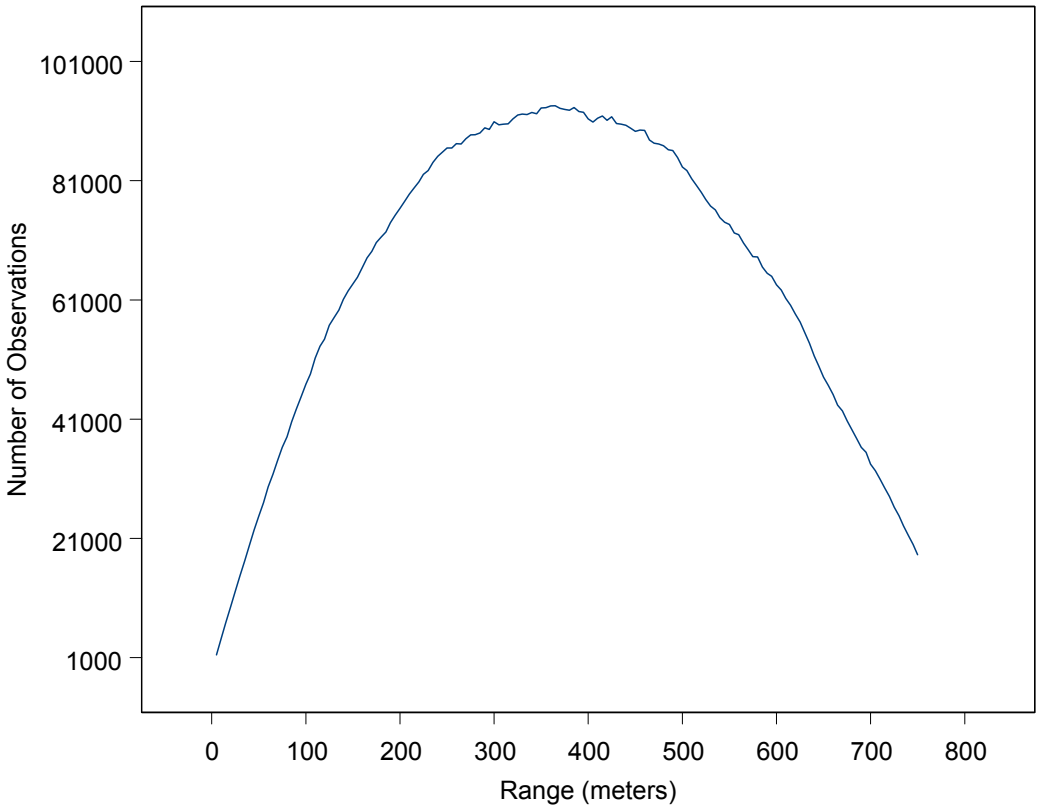


Figure 10. Illustration of the Number of Observations in Each Range Bin

a. City Core

For the City Core urban template (shown in Figure 5), we used the UPPS to calculate PLOS estimates for sensor elevations ranging from 0 to 2500 meters (target elevation was zero for all cases). The plot in Figure 11 shows a sampling of 10 PLOS curves that were generated for the City Core urban template.

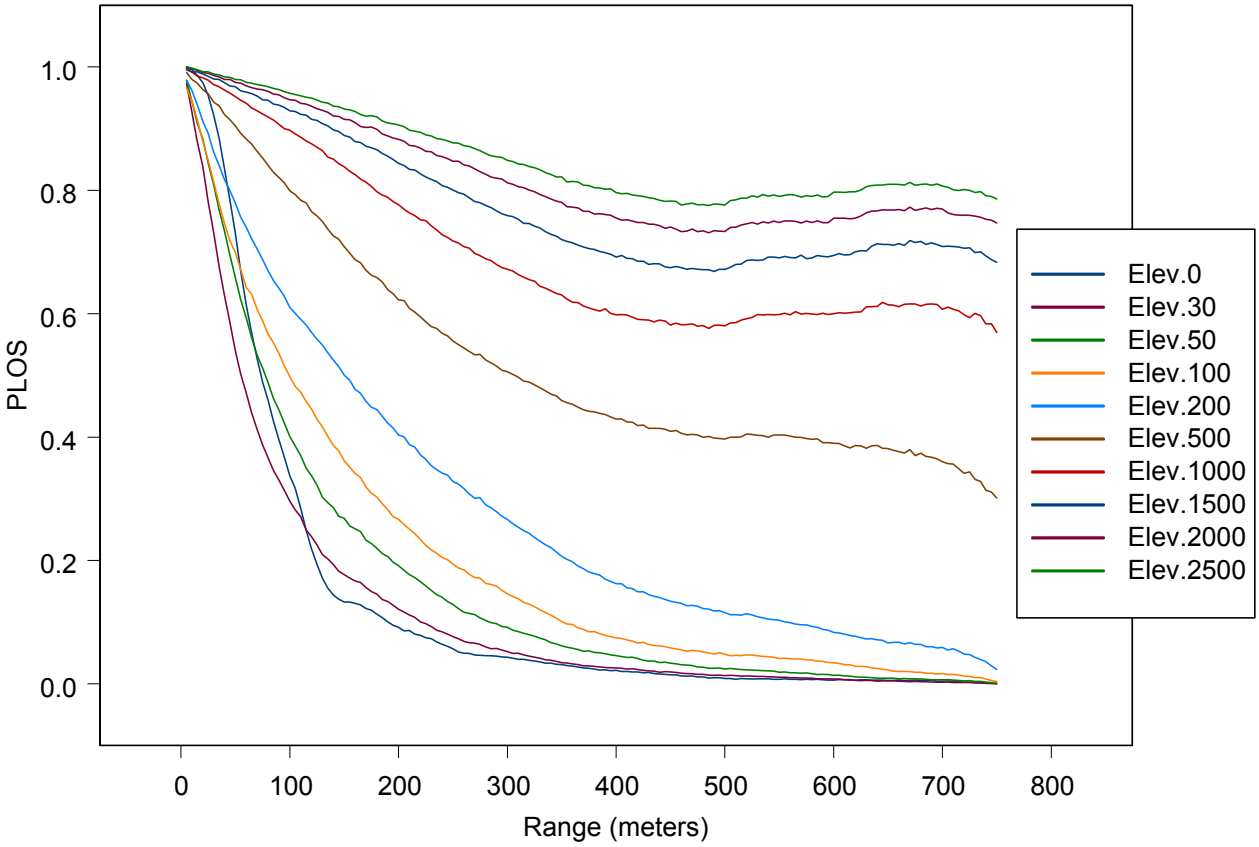


Figure 11. PLOS Curves for 10 Sensor Elevations in the City Core Urban Template

We now present three observations concerning the City Core PLOS curves depicted in Figure 11. First, note that the PLOS curve for zero elevation is higher than PLOS curves for higher sensor elevations for low ranges. In other words, this indicates that a sensor has a higher probability of having LOS to a target when it is located on the ground than when it is above ground. This conclusion seems counter-intuitive, but we have devised a theory that might explain this strange occurrence. Initially, we note that the PLOS curve for a sensor elevation of zero will be identical to that of any PLOS curve for a sensor elevation lower than the height of the shortest building. Figure 12 will help to explain this concept. For any sensor-target location combination (where the target elevation is zero), the evaluation of LOS will be same for any sensor elevation lower than the height of the shortest building (the same buildings will always impede, or not impede, LOS). Thus, with all of the LOS evaluations being the same, the resulting PLOS estimates will be the same for any sensor elevation lower than the height of the shortest building.

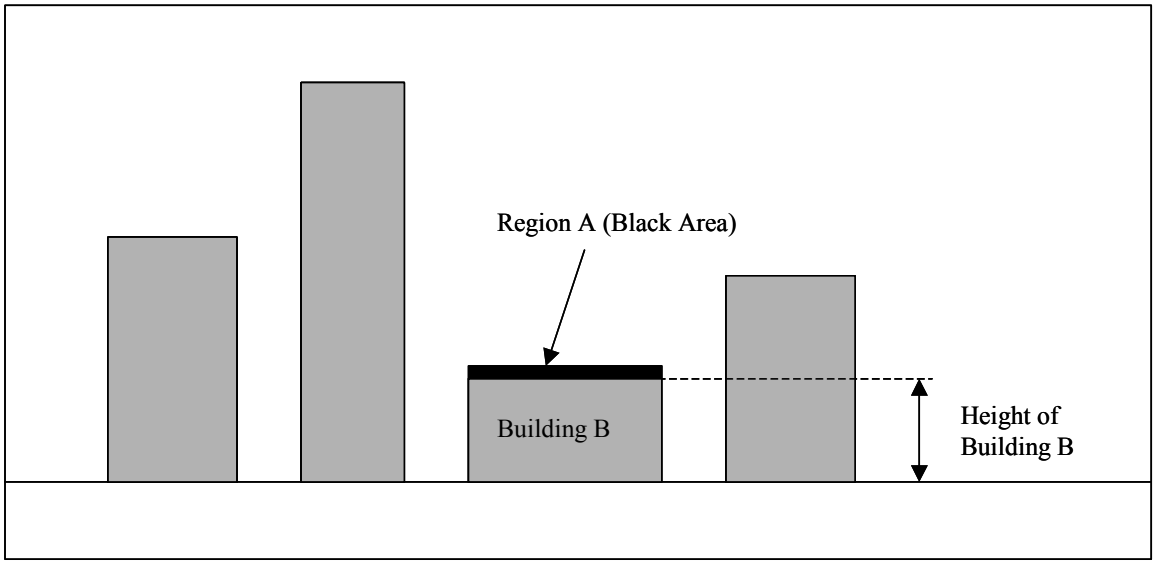


Figure 12. Illustration of Why PLOS Might Increase as Sensor Elevation Decreases

Now, consider moving the sensor elevation from an elevation lower than the height of the shortest building to an elevation slightly above the height of the shortest building. Looking at Figure 12, we can see that elevating the sensor above the height of the shortest building has introduced more sensor locations to consider; specifically, those sensor locations in region A of Figure 12. The sensor locations in region A have a low PLOS, since building B effectively blocks LOS to most ground-level targets. Thus, these additional sensor locations allowed by the increase in sensor elevation may decrease the total PLOS estimate, potentially making it worse than for a lower sensor elevation.

A second observation is that the PLOS curves for high (relative) sensor elevations actually *increase* as range increases on certain portions of the curve. This is completely counter-intuitive – we would expect that as the sensor-to-target range increases, the PLOS value would decrease, not increase. While we have not done thorough testing to evaluate this issue, we have again developed a theory to explain this phenomenon. Consider a sensor located directly above a building, as depicted in Figure 13. This sensor does not have LOS to targets that are within a range of r. However, when the range to the target is greater than r, the sensor will have LOS to the target. Thus, increasing the range from sensor to target in this case has *improved* the chance that the sensor has LOS to the target. This simple illustration can help explain why PLOS might increase as range increases.

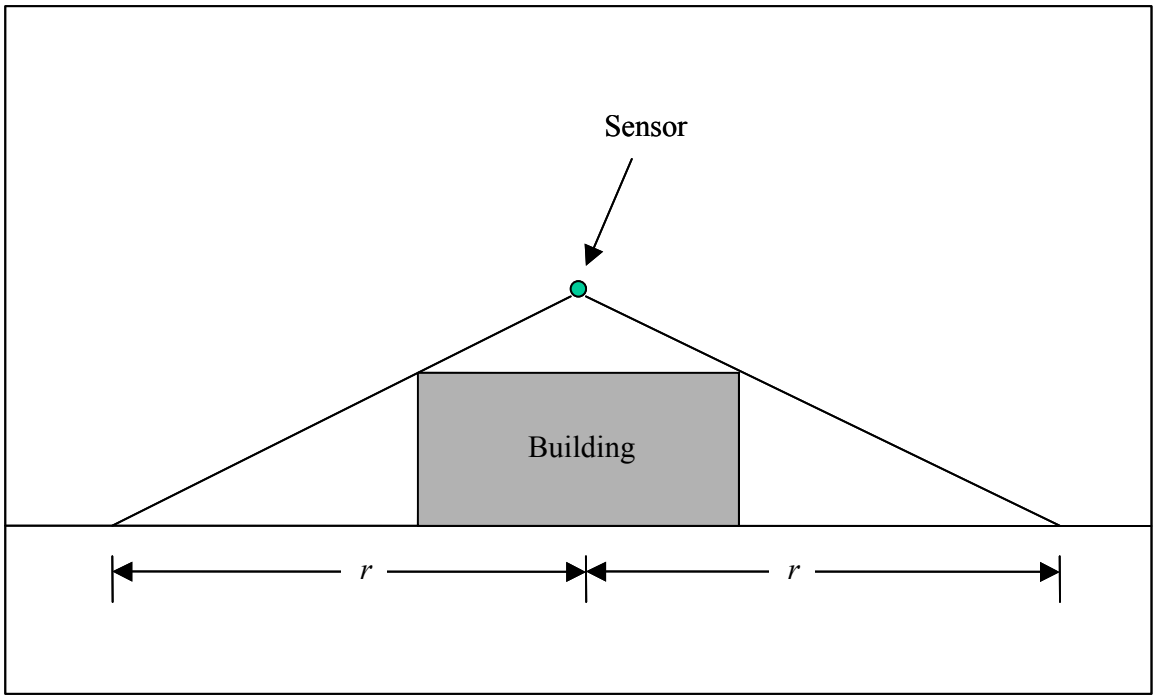


Figure 13. Illustration of Why PLOS May Increase as Range Increases

A final observation is that the PLOS curves in Figure 11 seem to have different shapes for low, medium, and high sensor elevations. This would suggest that when trying to approximate these curves with an analytical function, one might consider using three different types of analytical functions to approximate – one for low sensor elevations (lower than the height of the shortest building), one for medium sensor elevations (within the range of building heights – higher than the shortest building, but lower than the tallest building), and one for high sensor elevations (higher than the tallest building).

b. Outlying High-Rise Area

The plot in Figure 14 shows a sampling of 10 PLOS curves that were generated for the Outlying High-Rise Area urban template (shown in Figure 6). Note that these are the same sensor elevations as for the City Core urban template.

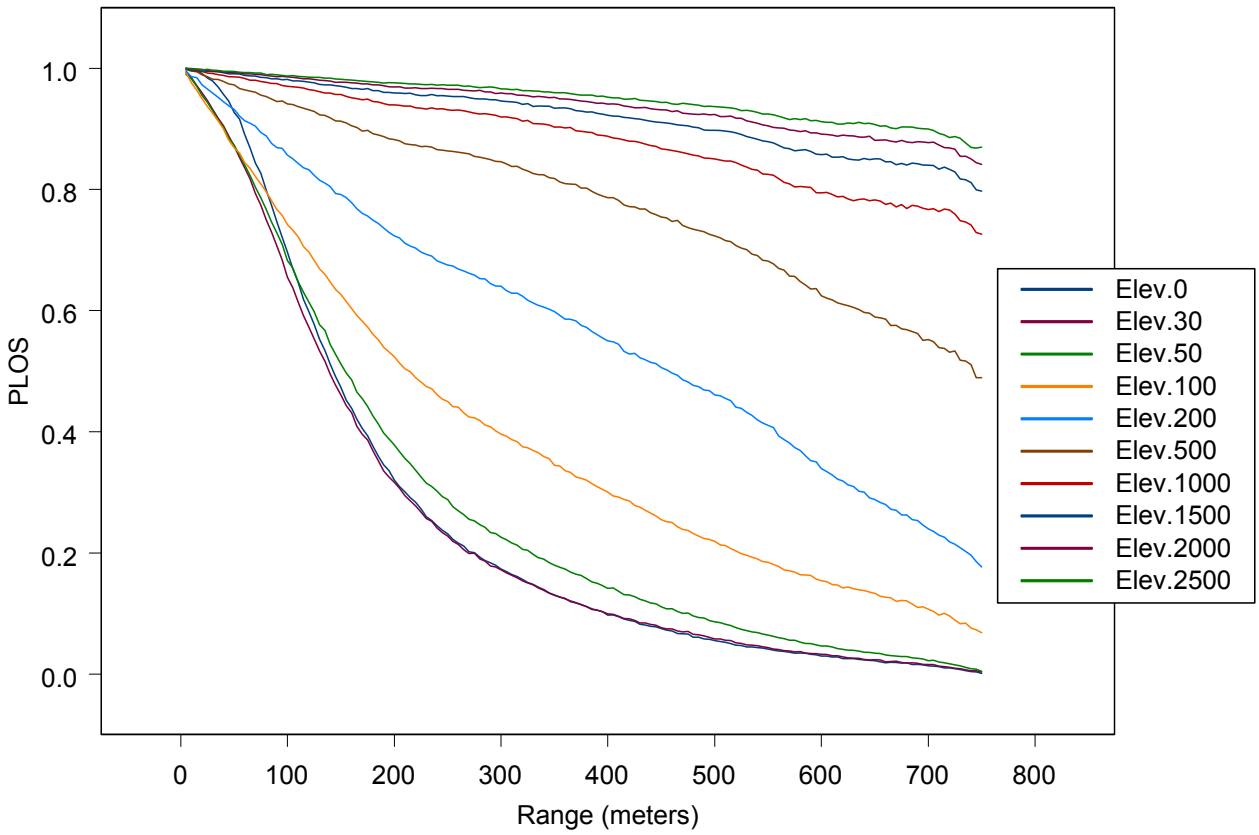


Figure 14. PLOS Curves for 10 Sensor Elevations in the Outlying High-Rise Area Urban Template

We now present three observations concerning the Outlying High-Rise Area PLOS curves depicted in Figure 14. First, just as with the City Core PLOS curves, we again find that the PLOS curve for zero elevation is higher than PLOS curves for higher sensor elevations for low ranges. The same explanation for why this happened with our City Core PLOS curves may explain why this happens for the Outlying High Rise Area PLOS curves, as well. Second, we note that, similar to the City Core PLOS curves, the Outlying High-Rise Area PLOS curves seem to have different shapes for low, medium, and high sensor elevations – this would suggest using different types of analytical functions in order to approximate these curves. Finally, we note that the PLOS curves for high (relative) sensor elevations seem to be almost linear over the set of ranges that we tested.

c. Commercial Ribbon

The plot in Figure 15 shows a sampling of 12 PLOS curves that were generated for the Commercial Ribbon urban template (shown in Figure 7). The sensor elevations used to construct these curves were different from those used for the City Core and Outlying High Rise Area urban templates. The Commercial Ribbon urban template had much shorter buildings; thus, it was appropriate that we focus our attention on a "lower" set of sensor elevations.

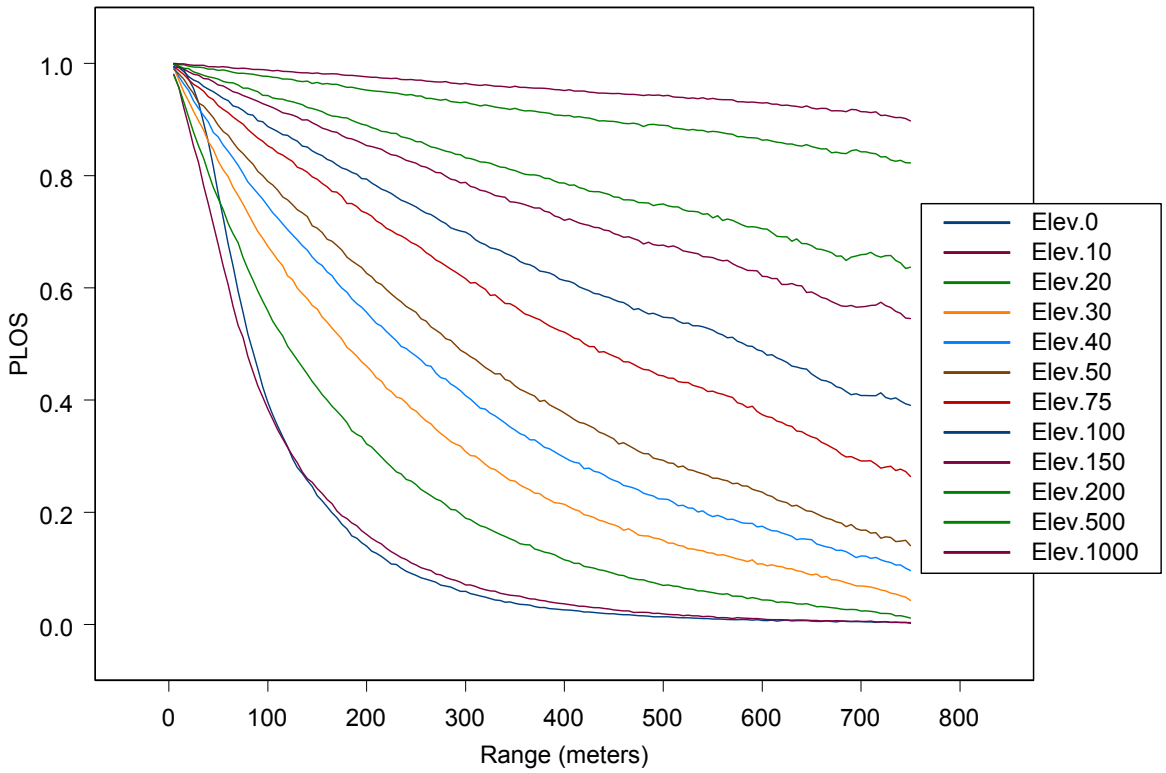


Figure 15. PLOS Curves for 12 Sensor Elevations in the Commercial Ribbon Urban Template

We now present three observations concerning the Commercial Ribbon PLOS curves depicted in Figure 15. First, just as with the City Core and Outlying High-Rise Area PLOS curves, we again find that the PLOS curve for zero elevation is higher than PLOS curves for higher sensor elevations for low ranges. Second, we note once more that the Commercial Ribbon PLOS curves seem to have different shapes for low, medium, and high sensor elevations and might require different classes of functions to approximate. Finally, as with the Outlying High-Rise Area PLOS curves, we see that the Commercial Ribbon PLOS curves for high (relative) sensor elevations seem to be almost linear over the set of ranges that we tested.

d. Comparison of PLOS Curves For All Three Urban Templates

The next logical step in our analysis was a simple comparison of PLOS curves with identical sensor elevations but different urban template classifications. Figure 16 shows the PLOS curve of each urban template for a 50-meter sensor elevation (left) and a 200-meter sensor elevation (right). Not surprisingly, these plots demonstrate that the Commercial Ribbon urban template is the "dominant" urban template for LOS – it has a higher PLOS estimate for any given range than do the other two urban templates. Also, as we might expect, the City Core urban template is the "worst" urban template for LOS. Similar plots to those in Figure 16 for different sensor elevations yielded the same ordinal results.

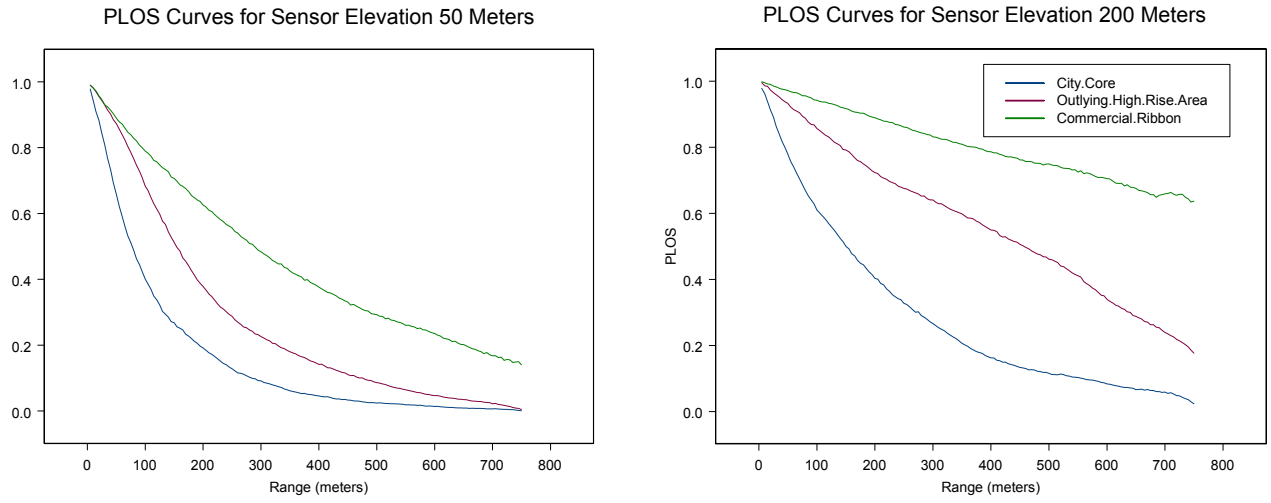


Figure 16. PLOS Curves for Each Urban Template for Sensor Elevation 50 Meters (left) and 200 Meters (right)

2. Comparison of UPPS PLOS Estimates to PLOS Estimates Generated With Combat XXI

Generating urban PLOS estimates with the UPPS was certainly an accomplishment, but how could we be certain that these urban PLOS estimates were accurate? We wanted a "second opinion" that, at least on a superficial level, would provide us with some confidence that the urban PLOS estimates produced by the UPPS were precise. We found an existing entity-level combat simulation, Combat XXI, to be useful in obtaining this "second opinion."

Combat XXI is a replacement and upgrade for Combined Arms and Support Task Evaluation Model (CASTFOREM), the Army's highest resolution, lowest echelon, systemic, combined arms combat simulation model (Army MSRR, 2004). Combat XXI is capable of representing urban terrain and has its own LOS algorithm in order to evaluate battlefield encounters. We used existing Combat XXI code (with a few modifications) and its LOS algorithm to generate PLOS estimates for some of the same ranges, sensor elevations, and terrain types as we did with the UPPS. The code used to generate these urban PLOS estimates with Combat XXI can be found in Appendix C. As an example of our results, Figure 17 shows a comparison of the PLOS estimates generated by the UPPS and Combat XXI for the Commercial Ribbon and Outlying High-Rise Area urban templates with sensor elevations of 100 and 200 meters, respectively. (We performed many other comparison plots for different ranges, sensor elevations, and terrain types; however, we do not display them in this paper.)

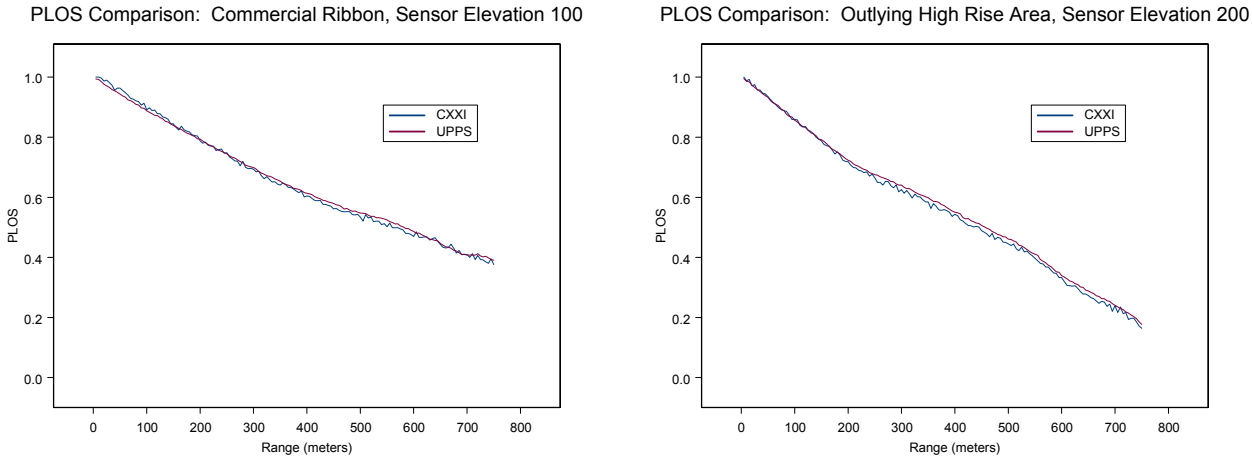


Figure 17. Comparison of PLOS Estimates Generated By the UPPS and Combat XXI

The similarity of the PLOS estimates that we computed using Combat XXI to the PLOS estimates computed with the UPPS should in no way be considered a validation of the UPPS; instead, they should simply be considered a "sanity check" for verifying the accuracy of the PLOS estimates generated by the UPPS. Nonetheless, the PLOS estimates computed using Combat XXI are at least a first step in the validation of the UPPS. In that respect, it is encouraging to note that the two sets of PLOS estimates computed (the UPPS and Combat XXI) never differed by more than 0.08 for any of the ranges, sensor elevations, and terrain types that we tested. The exact reason for the differences the two sets of PLOS estimates is unclear at this point in our work; this is a subject to which further research should be dedicated.

3. Effect of Change in Target Elevation

Once we were successful at generating PLOS curves with ground-level targets, it was natural to extend the UPPS to handle elevated targets. The UPPS was modified to allow the user to vary the target elevation while keeping the sensor elevation fixed. For example, one can generate PLOS curves for each combination of sensor elevation and target elevation. Specifically, this would allow one to explore PLOS curves for rooftop targets. It should be noted, however, that the introduction of target elevation necessitates the use of an extra parameter in our construction of PLOS curves; PLOS curves generated for a specific target elevation require four parameters: range, sensor elevation, terrain type, and target elevation.

4. Effect of Underlying Terrain Skin

In developing the UPPS, we made the assumption that only buildings could block LOS by omitting the underlying terrain skin. Now, the question arises: does underlying terrain skin make a substantial difference when calculating urban PLOS estimates? We enlisted the help of AMSAA and TerraSim to help us take the available geotypical urban templates and mount them on an underlying terrain skin. Figure 18 shows an example of the results of this "mounting" – it depicts the City Core urban template with an underlying terrain skin.

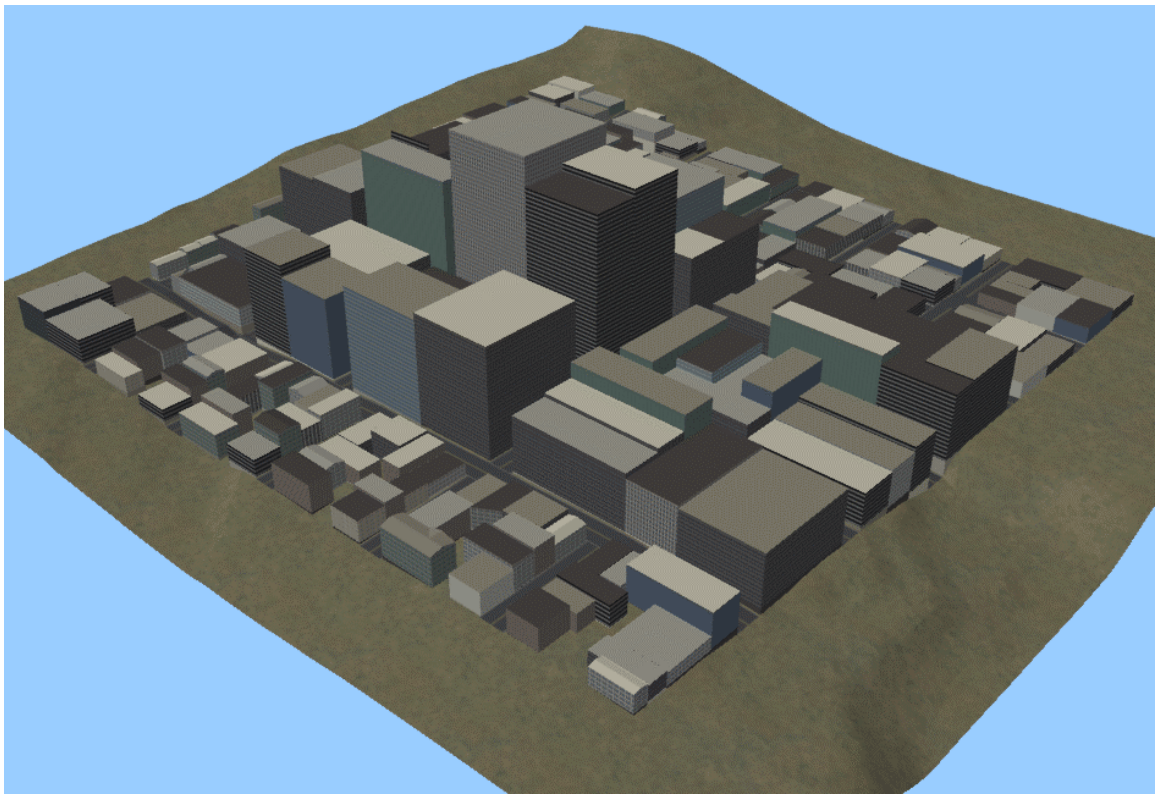


Figure 18. City Core Urban Template With an Underlying Terrain Skin (From ref. McKeown, 2003)

A full analysis of the effect of underlying terrain skin was not completed in this work; however we nonetheless suggest an experimental design by which one would study these effects. Specifically, one would construct pairs of PLOS curves for fixed sensor elevations in each of the three available urban templates. Of each pair of PLOS curves, one would be constructed omitting the underlying terrain skin (assuming a flat underlying surface); the second would be constructed using elevation data after mounting the urban template on a non-flat underlying terrain skin. Each pair of PLOS curves can then be statistically analyzed to determine if there are significant differences in PLOS estimates when including the underlying terrain skin.

5. Time Until LOS is Gained or Lost

In this research, our simulation runs focused on stationary sensors and targets. Thus, the urban PLOS estimates that we have generated using the UPPS are only valid for a particular point in time. As an extension, we explored how we could use our urban PLOS estimates to approximate the time until LOS is gained or lost for moving targets in an urban environment. Specifically, we accomplished this by deriving a cumulative distribution function (CDF) for the time until LOS is gained and lost for a particular sensor elevation and urban terrain type.

a. Assumptions

In deriving our CDFs, we made the following assumptions:

1. Sensor is stationary (thus, its elevation is fixed).
2. Urban terrain type is fixed.
3. The target is moving (thus, its range to the sensor is changing).
4. Successive LOS "attempts" by the sensor are independent.

b. Variables

T_G	time until LOS is gained
T_L	time until LOS is lost
r_0	initial range from sensor to target
$r(t)$	change in sensor-target range in a time of t
R	set of ranges at which sensor-target LOS is evaluated (note: this is an infinite set of ranges – it includes all ranges in the interval $[r_0, r_0 + r(t)]$)
$PLOS(r)$	PLOS for sensor-target range r (remember, urban terrain type and sensor elevation are already fixed)

c. CDF for the Time Until LOS is Gained

$$
\begin{aligned}
P\{T_L \le t\} &= 1 - P\{T_L > t\} \\
&= 1 - P\{\text{LOS is never gained during time } t\} \\
&= 1 - P\{\text{LOS is never gained along the target path}\} \\
&= 1 - \prod_{r \in R} \left(1 - PLOS(r)\right) \\
&= 1 - \exp\left\{ \ln\left\{ \prod_{r \in R} \left(1 - PLOS(r)\right) \right\} \right\} \\
&= 1 - \exp\left\{ \sum_{r \in R} \ln\left\{ \left(1 - PLOS(r)\right) \right\} \right\} \\
&= 1 - \exp\left\{ \int_R \ln\left\{ \left(1 - PLOS(r)\right) \right\} dr \right\}.
\end{aligned}
$$

d. CDF for the Time Until LOS is Lost

$$
\begin{aligned}
P\{T_L \le t\} &= 1 - P\{T_L > t\} \\
&= 1 - P\{\text{LOS is maintained during time } t\} \\
&= 1 - P\{\text{LOS is maintained along the target path}\} \\
&= 1 - \prod_{r \in R} PLOS(r) \\
&= 1 - \exp\left\{ \ln\left\{ \prod_{r \in R} PLOS(r) \right\} \right\} \\
&= 1 - \exp\left\{ \sum_{r \in R} \ln\left\{ PLOS(r) \right\} \right\} \\
&= 1 - \exp\left\{ \int_R \ln\left\{ PLOS(r) \right\} dr \right\}.
\end{aligned}
$$

e. Use of CDFs in a Combat Simulation

Just as our PLOS estimates can be calculated during the pre-processing phase of a combat simulation, so can the estimates proposed by the CDFs for the time until LOS is gained or lost. Since the integrals in the equations of these CDFs are unlikely to have a closed form solution, a simulation would need to use a numerical approximation to evaluate these integrals; specific methods for approximating definite integrals numerically can be found in (Gerald, 1999). Once these CDFs are generated during preprocessing, a simple uniform random number draw and comparison to the CDF could be used to schedule the time when LOS is gained or lost in a discrete event simulation. Additionally, using these CDFs, it would be easy to simulate a target that alternates between being visible and hidden. Once LOS to a target is gained, the time until LOS is lost could be compared to times required for a sensor to classify/detect/identify a target to determine if the target has been visible long enough to be placed into a "higher" acquisition state.

III. STOCHASTIC AND ANALYTICAL MODELS FOR TARGET ACQUISTION IN URBAN TERRAIN

A. BACKGROUND

Many current combat simulations model urban target acquisition at the entity-level. These models are extremely time-intensive, as they evaluate the target acquisition outcome of each sensor-target combination on the battlefield. Combine this with the fact that, in general, these target acquisition models use physics-based and/or computationally expensive algorithms to determine the target acquisition outcome for each sensor-target encounter, and the result is a combat simulation that can take a long time to execute for large-scale urban combat scenarios. With an important aspect of current and future military operations being in an urban setting, entity-level target acquisition models will only become more cumbersome.

The complexities encountered in urban target acquisition modeling largely parallel those of attrition modeling. When approached from the entity-level, attrition modeling is also extremely time-intensive and can become computationally expensive in a combat simulation. However, combat simulation developers have found ways to simplify calculations of attrition, and, in the process, have improved the run time of their simulations. Specifically, some current combat simulations, such as Combat XXI and CASTFOREM, use probabilities of hit and probabilities of kill in place of computationally expensive algorithms of attrition modeling. For example, instead of a combat simulation using a physics-based algorithm to determine if a round fired from a friendly tank hits an enemy tank (an algorithm that might use, for example, round trajectory, weather conditions, and obscuration effects in its calculations), the simulation simply uses a probability of hit and/or probability of kill to determine the extent of the damage caused by the fired round. Probabilities of hit and kill are determined in pre-processing and are subsequently accessed during the simulation either via a table look-up or analytical function evaluation based on a few predetermined parameters, such as range, round type, and target type.

B. OBJECTIVES

The objective of this research is to develop stochastic and analytical target acquisition models whose outputs could be used in place of computationally expensive urban target acquisition algorithms in existing and emerging combat simulations to provide for faster simulation time while yielding an acceptable compromise in accuracy on the aggregate level. In a sense, we are seeking the urban target acquisition equivalent to probabilities of hit and probabilities of kill used in attrition modeling.

Specifically, we aim to develop mathematical models for aggregate urban target acquisition that would use data derived from high resolution simulations as inputs in order to rapidly estimate desired quantities and, in turn, provide insight into the following areas:

1. Estimating the proficiency of a given set of sensors against a given set of targets in a specific urban area.

2. Determining the number and mix of sensors required to achieve a certain level of "dominance" in a particular urban area.

3. Providing suggested employment strategies for sensors.

C. PROPOSED MODELS

1. Cumulative Distribution Function Model

The goal of this model is to estimate the proficiency of a given set of sensors against a given set of targets in a particular urban area. In this section, the term "detection" is used to generically represent any level of target acquisition. It can be replaced everywhere by another type of target classification, say, "recognition" or "identification," and similar models would apply.

a. Problem Description

A large number of sensors (the "blue" force) are arrayed against a large number of targets (the "red" force) in a particular urban Area of Operations (AO). The

blue force knows the number and type of sensors it has available, but does not know the number and type of targets that exist in the AO.

b. Model Objectives

Our model is designed to answer the following questions:

1. How many (and which) targets are detected by a given set of sensors in a given time period?

2. How many (and what type of) sensors will it take to detect a certain percentage of targets in a given time period?

c. Approach

Many existing models used in current combat simulations are certainly capable of answering these questions; however, they would often use time-intensive, computationally expensive algorithms to do so. Our approach to answering the two questions above is radically different: we develop an aggregate target acquisition model that significantly reduces the number of calculations required to estimate desired quantities. Specifically, rather than seeking a deterministic answer to the above questions, we instead pursue a stochastic answer. For example, to answer Question #1, we will derive a general expression that describes the *distribution* of the number and type of targets detected (from which we can generate the number and type of targets detected for any particular instance).

d. Model Development

Assume we have a large number of blue force sensors arrayed against a large number of red force targets in a particular AO. The blue force is comprised of a set of sensor packages, S, each consisting of one or more sensor platforms. Each sensor platform, in turn, consists of different types of sensor assets. A concise description of the relationship between individual sensors, sensor platforms, and sensor packages is given

in (Tutton, 2003). Each red force target is categorized as being of a particular type; we let T be the set of target types. Figure 19 shows an example of a complete set of sensors and targets in our model.

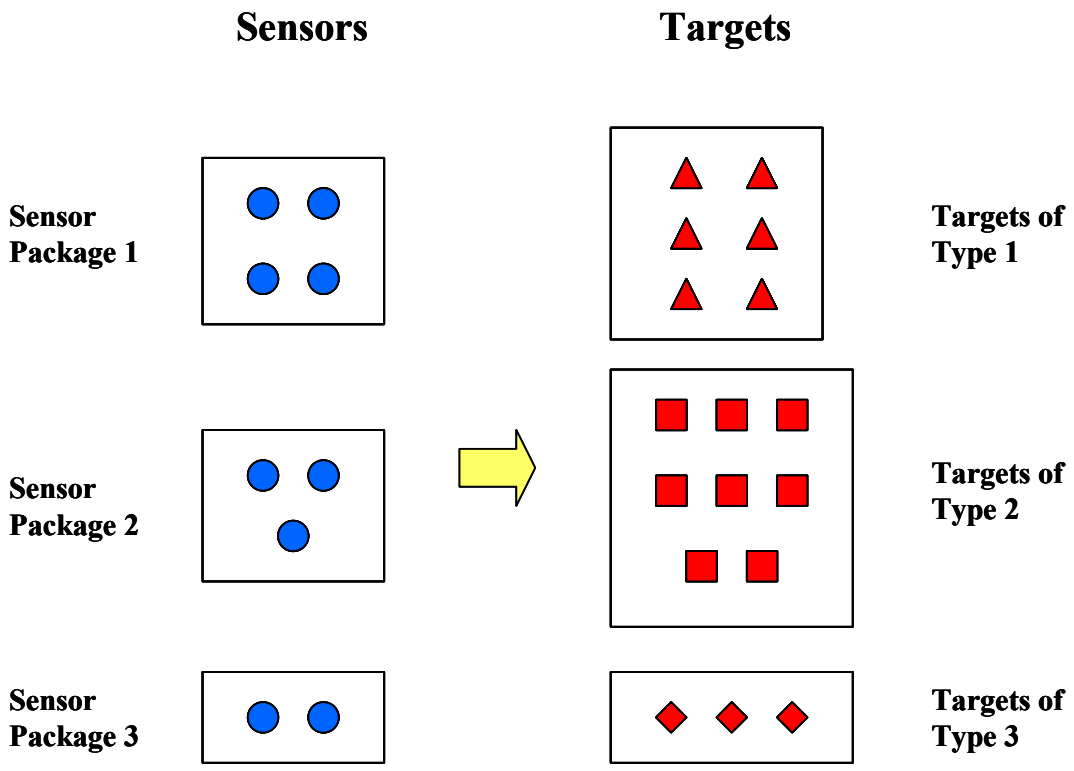
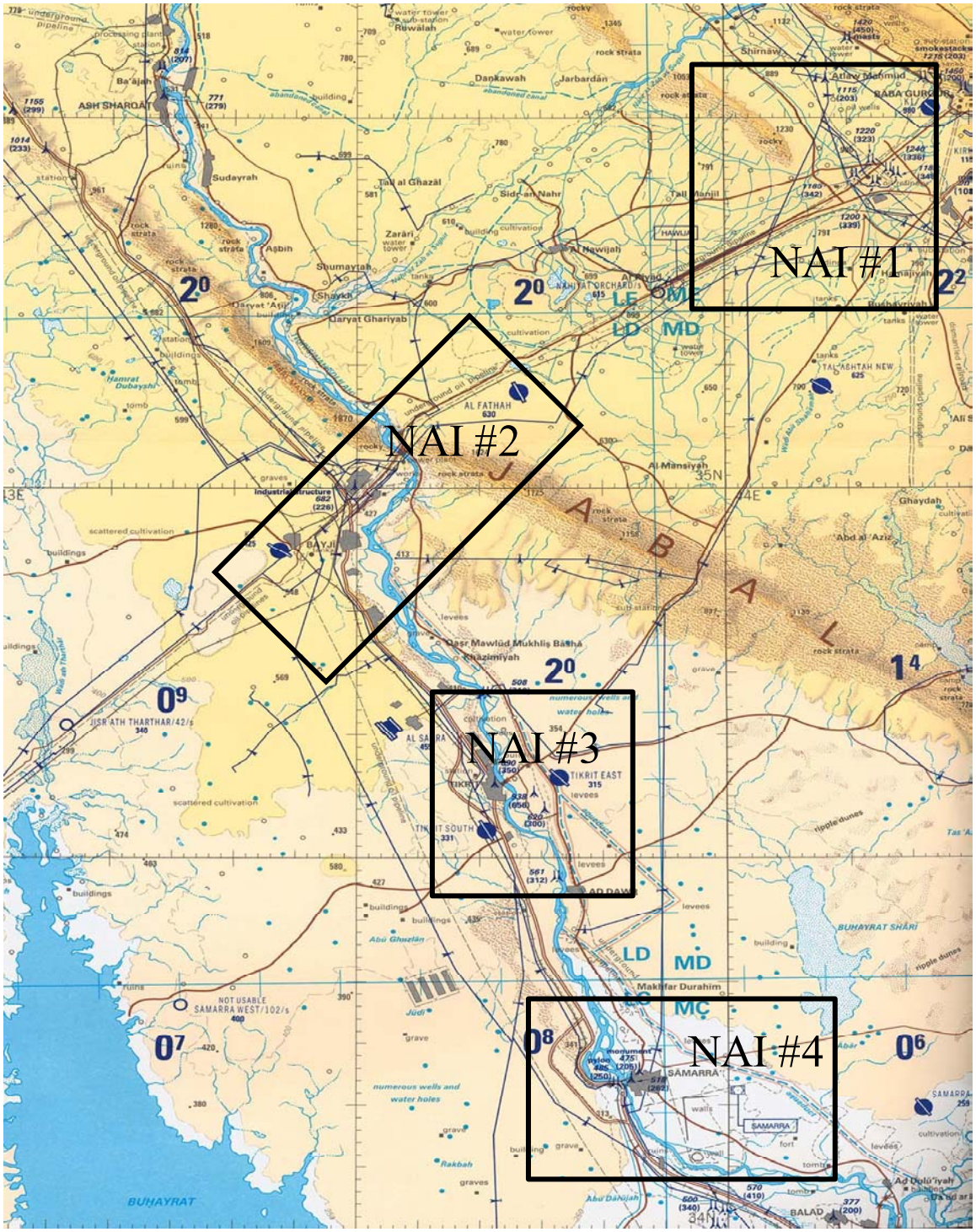


Figure 19. An Example Set of Sensors and Targets for the Cumulative Distribution Function Model

In conducting intelligence-gathering operations prior to battle, the blue force will begin by partitioning the AO into n named areas of interest (NAIs) that will be searched by its sensor assets. We will assume that the NAIs do not exhaust the entire AO. Now, since the blue force does not have complete knowledge of the red force, there will certainly be targets that are not contained in any NAI. Thus, the set of targets will effectively be partitioned into n+1 target clusters (n clusters of the set of targets contained in each of the NAIs and one cluster of the set of targets that are not contained

in any NAI). We will call this set of target clusters C. We assume, in this paper, that a decision tool exists and is used by the blue force commander for partitioning the area of operations into NAIs (and, thus, the set of targets into clusters). An example of how an AO might be partitioned into NAIs is given in Figure 20.



In the AO pictured above, there are 4 NAIs (bounded by the rectangles). Thus, the set of targets would be partitioned into 5 target clusters.

Figure 20. An Example of How an AO Might be Partitioned Into NAIs (From ref. Tutton, 2003)

Now, there are $|C|$ target clusters that the blue force has classified as "eligible" for search by its sensor assets (we may assume that $|C| > |S|$, in general). We must also assume, in a worst case, that each target cluster would contain all $|T|$ different types of targets. Figure 21 summarizes our problem by removing the "map" containing the AO and NAIs and depicting only sensors and targets.

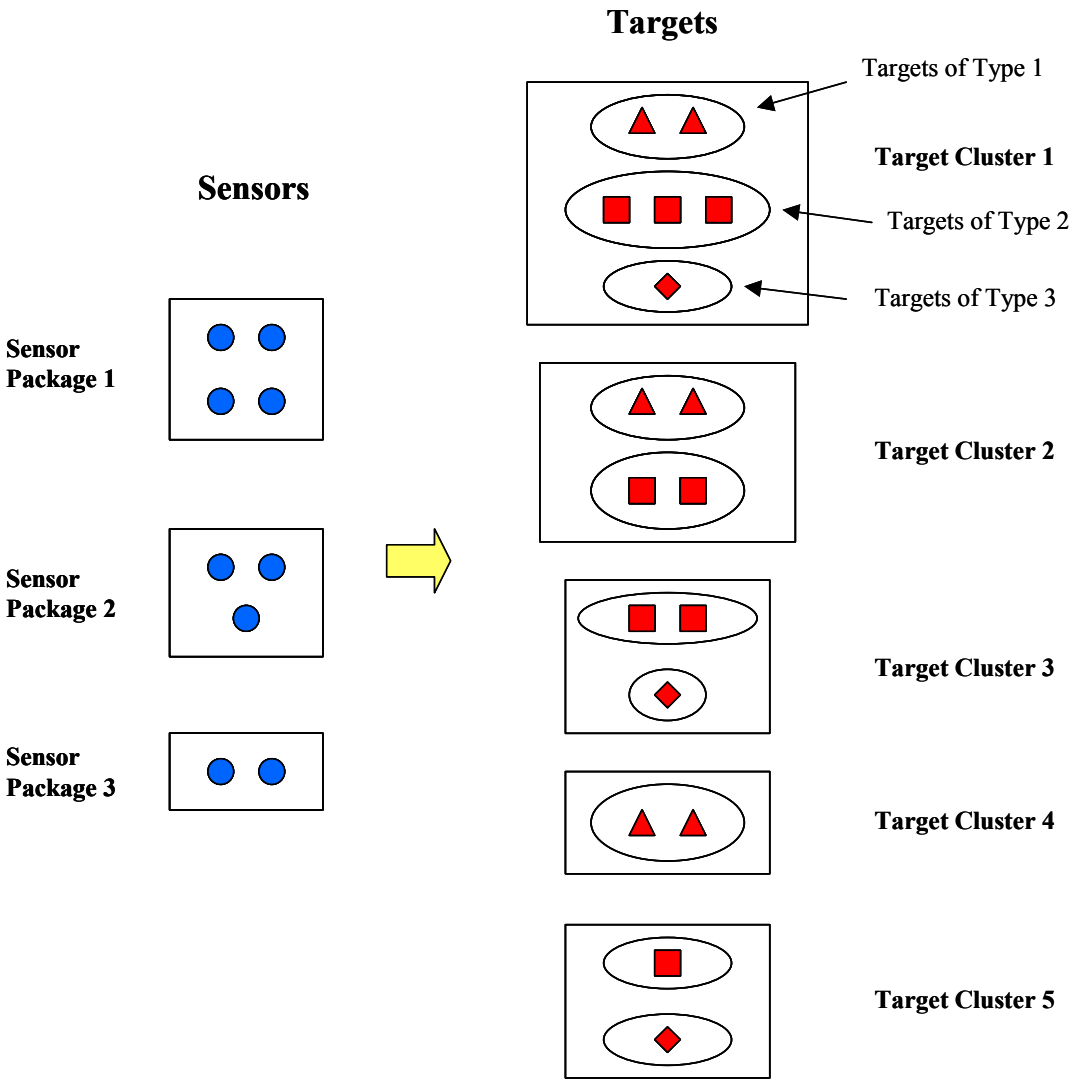


Figure 21. Partitioning the Set of Targets into Target Clusters

The blue force will assign its sensor packages to search each NAI (each NAI will have at most one sensor package assigned to search it). We again assume that a decision tool exists and is used by the blue force for optimally assigning sensor packages to NAIs; an example of such a decision tool is given in (Tutton, 2003). Figure 22 elaborates on Figure 21, exhibiting an allocation of sensor packages to target clusters.

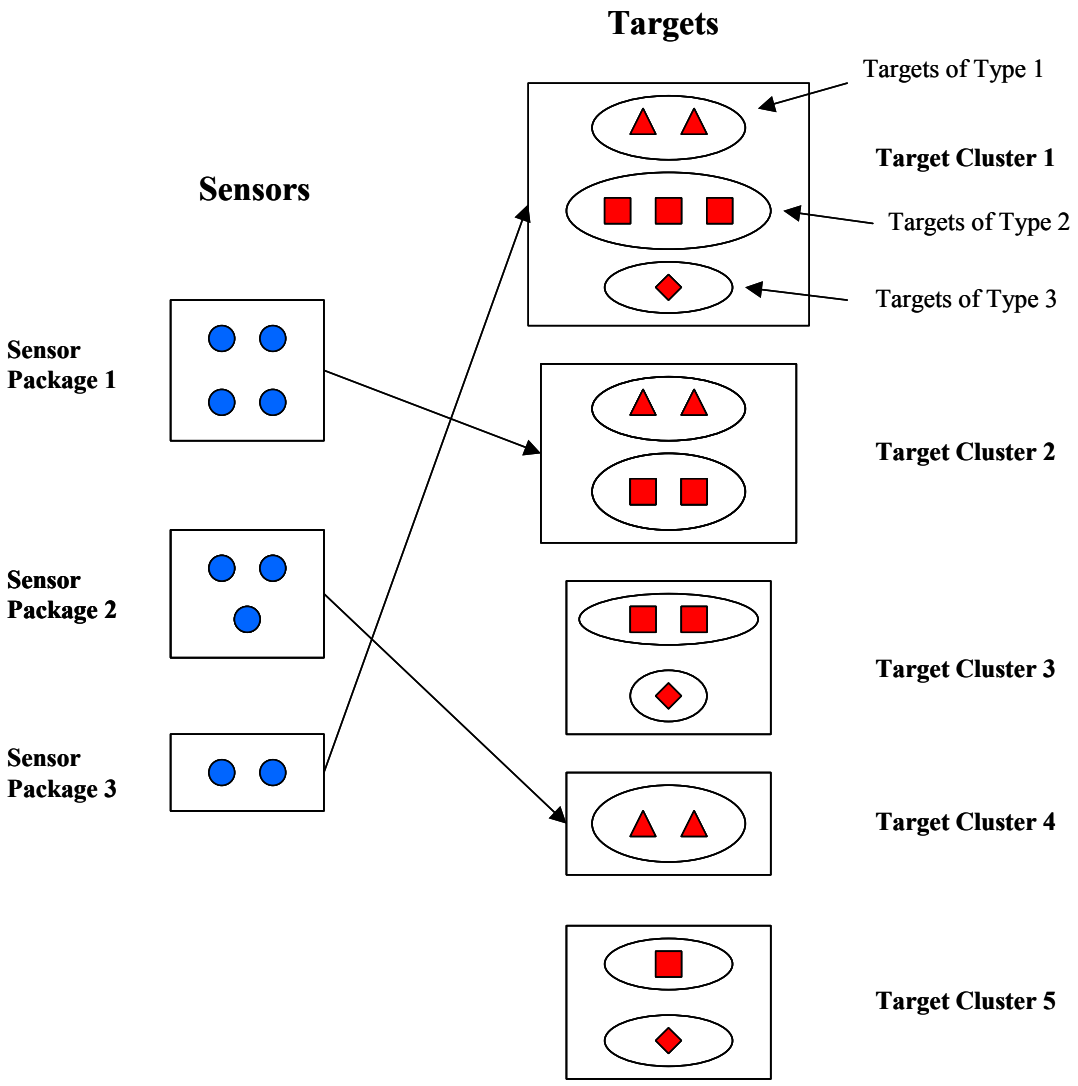


Figure 22. An Allocation of Sensor Packages to Target Clusters

Now, each sensor package will be searching for potentially $|T|$ different target types. Thus, the allocation of sensor packages to NAIs effectively divides our problem into $|S| \cdot |T|$ subproblems (we have $|S| \cdot |T|$ sensor package/target type combinations to investigate). We will focus our attention on one of these general subproblems, depicted in Figure 23. We will first derive an answer to the following question: how many targets of type *t* in target cluster *c* are detected by sensor package *s* in a given time period? We will label the answer to this question as X_{tcs}.

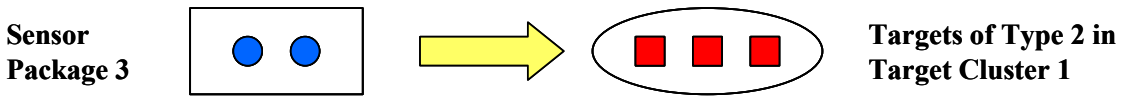


Figure 23. A Subproblem in the Cumulative Distribution Function Model

Again, rather than seeking a deterministic answer to the above question, we instead pursue a stochastic answer. Specifically, we will derive a cumulative distribution function (CDF) that describes the distribution of how many targets are detected. Then, for any particular subproblem, a simple random number draw and comparison to the CDF will generate a value of X_{tcs}. (This explains why we have used a capital letter X to label this quantity – it is a random variable in our model.)

Rather than assuming that each sensor acts independently of the other sensors in sensor package *s*, we instead assume that we can aggregate the sensors in sensor package *s* and effectively treat them as a single sensor (the manner in which this is done is at the discretion of the modeler; specific examples can be found in (Tutton, 2003) and (Driels, 2001). Thus, we can think of sensor package *s* (now considered a single, aggregated sensor) as having the opportunity to detect each target in the NAI to which it is assigned. Let y_{tc} be the number of targets of type *t* in target cluster *c*. We can view

the search by sensor package s for targets of type t in target cluster c as y_{tc} independent searches for targets.

We are now ready to begin calculation. Let p_{tcs} be the probability that a target of type t in target cluster c is detected by sensor package s in a given time period (this time period should be the same for all p_{tcs}). We can assume that we are able to calculate these probabilities of detection for each target type/target cluster/sensor package combination (or that these probabilities of detection have already been calculated and can be referenced in a table) and that the same target type/target cluster/sensor package combination will always yield the same probability of detection. The way in which these probabilities of detection are calculated is at the discretion of the modeler. Parameters that might be used to calculate these probabilities of detection are (1) the search time available to the sensor package, (2) the size of the NAI, (3) the "effectiveness" of the sensor package, (4) the number and type of sensors in the package, (5) the number and type of targets in the target cluster. For an example of how these probabilities might be calculated for a given sensor package and target cluster, see (Tutton, 2003) or (Driels, 2001).

Again, our aggregation model assumes that the values of p_{tcs} can be calculated; *it does not depend on how they were calculated*. Figure 24 shows these probabilities of detection applied to our model.

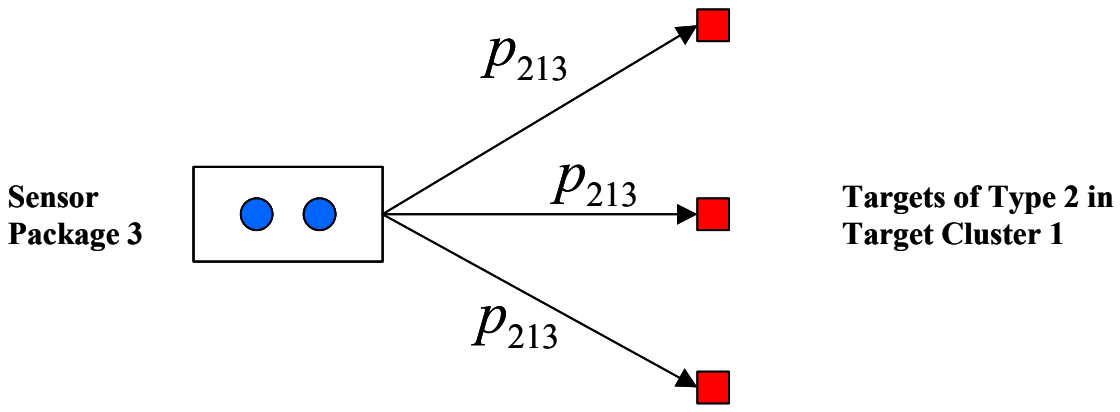


Figure 24. Probabilities of Detection Applied to the Cumulative Distribution Function Model

Once a probability of detection is assigned to each target, we can perform a binomial probability calculation to determine the probability of detecting any number of targets of each type. Specifically, we can say that

$$P\{X_{tcs} = x\} = \binom{y_{tc}}{x} p_{tcs}^{x} \left(1 - p_{tcs}\right)^{y_{tc} - x}.$$

This probability distribution can be transformed into a CDF using the following expression:

$$P\{X_{tcs} \le y\} = \sum_{x \le y} P\{X_{tcs} = x\}.$$

Thus, in order to generate a particular value of X_{tcs}, we would draw a uniform random number in the interval $[0,1)$ – label this random number *u*. Next, we would compare *u* to the values of our CDF; the smallest value of *x* for which $P\{X_{tcs} \le x\} > u$ will be our generated value of X_{tcs} (the number of targets of type *t* in target cluster *c* detected by sensor package *s* in a given time period).

Now, having generated X_{tcs}, we can address another question: *which* targets of type *t* in target cluster *c* are detected by the sensor package *s* in the given time period? Since we have assumed that all sensor package/target type encounters in the same NAI would be assigned the same probability of detection, we can answer this question by simply randomly selecting X_{tcs} targets from the y_{tc} targets of type *t* in cluster *c*.

We can perform the same calculations above for every target type/target cluster/sensor package combination, sequentially deriving CDFs ($|S| \cdot |T|$ of them) from which we can generate values of X_{tcs} and, in turn, randomly generate *which* targets are detected. This procedure will precisely answer our first question: how many (and which) targets are detected by a given set of sensors in a given time period?

Now, without knowledge of exactly how each sensor package was aggregated into a single sensor, we cannot easily determine how many and what type of sensors are required to detect a certain percentage of targets (our second question). Instead, we propose a method by which the blue force can heuristically analyze several courses of action (the number of type of sensors employed) and decide which is the best to fit its mission requirements.

The blue force would first need to decide on a decision criterion (or measure of effectiveness (MOE)) that will be used to decide whether or not a certain set of sensors is "better" than another for a given scenario. Next, the blue force can propose several "candidate" sets of sensors for use in the AO. Then, the model we have developed above (to answer our first question) could be applied for each candidate. The output values of each MOE could be statistically analyzed to provide a decision-maker with a summary of how well each set of sensors would perform. Thus, we have not answered our second question explicitly, but we have instead proposed a method by which the blue force can evaluate the strengths and weaknesses of several candidate sets of sensors quickly through the use of our existing model.

2. Target Acquisition State Transition Models

The goal of these models is to represent the "movement" of a target between different acquisition states. The model proposed in this paper uses four acquisition states: Undetected, Detected, Recognized, and Identified. Similar models could be developed for any number and names of acquisition states, though the complexity of the model will increase with the addition of each acquisition state.

A target is considered to be undetected if it is not detected at the present time; all targets in our models are always considered to be initially undetected. A target becomes detected when a sensor perceives an object of possible military interest that is unconfirmed by recognition (JP 1-02, 2001). The detected target becomes recognized only when it is determined that it is similar within a category of something already known; e.g. tank, truck, man (JP 1-02, 2001). In other words, if a detected target can be classified as being of a given category of objects, then it has been recognized. We note

here that, by definition, detection is a "prerequisite" for recognition. The recognized target, in turn, becomes identified only when a sensor is able to discriminate it as being friendly or enemy, and/or is able to assign a name that belongs to that target as a member of a class (JP 1-02, 2001). Again, we note here that, by definition, recognition is a prerequisite for identification.

a. Problem Description

An aggregated sensor package is searching for one target in a particular NAI representative of a specific urban terrain type.

b. Model Objectives

Our models are designed to answer the following questions:

1. What is the probability that a target becomes detected, recognized, or identified in a given amount of time?

2. How long does it take a target to transition from one acquisition state to another?

3. How much time will it take to detect, recognize, or identify a target?

c. Approach

Existing models used in current combat simulations are capable of answering these questions; however, they would likely use time-intensive, computationally expensive algorithms to do so. Our approach to answering the three questions above will be the same as for our Cumulative Distribution Function Model – rather than seeking a deterministic answer, we instead pursue a stochastic answer. For example, to answer Question #2, we will develop a model from which we can estimate

the *distribution* of transition times between acquisition states (from which we can generate a transition time for any particular instance). Additionally, our intent is to use data from high-resolution simulation runs in order to estimate desired quantities and ultimately answer the above questions.

We will develop two distinct models: (1) a Discrete Time Markov Chain Model, and (2) an Event Step Model. The goal of the Discrete Time Markov Chain Model will be to estimate the probabilities of transitioning from one acquisition state to another in a given time, while the goal of the Event Step Model will be to estimate the transition *times* between acquisition states.

d. Model Development – The Discrete Time Markov Chain Model

A Discrete Time Markov Chain is a stochastic process that takes on a finite or countable number of possible values. The possible values that the process may take are referred to as "states." Whenever the process is in state *i,* there is a fixed probability, P_{ij}, called a "single-step transition probability," that the process will next be in state *j* (this probability is conditionally independent of all past states the process has visited given that the process is in state *i* at the present time). Additional information on Discrete Time Markov Chains can be found in (Ross, 2003).

In particular, for our target acquisition problem, we have four states: Undetected, Detected, Recognized, and Identified. A target in any one of these states can transition to any other state. If we assume that the probability of a target moving from one acquisition state to another is conditionally independent of the past given its present state, then a Discrete Time Markov Chain can be developed to model the movement of a target through acquisition states. Figure 25 is a pictorial representation of this Discrete Time Markov Chain.

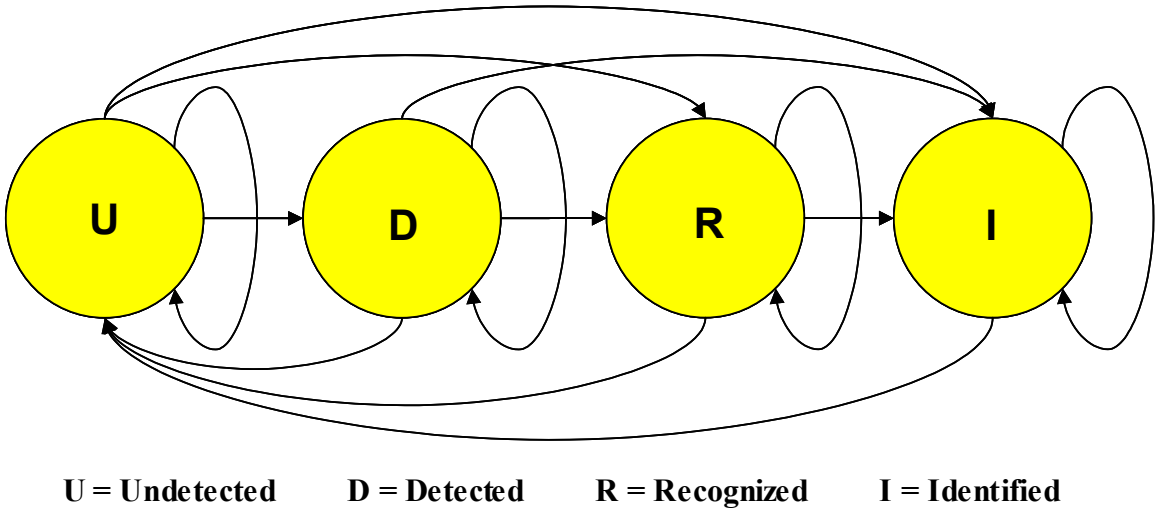


Figure 25. Discrete Time Markov Chain Model of Target Acquisition States

Each arc in Figure 25 has a single-step transition probability associated with it. Each single-step transition probability is the probability that the target moves along this arc (from the state at the "tail" of the arc to the state at the "head" of the arc) in the next time step. For example, P_{UD} is the probability that the target transitions from the Undetected state to the Detected state in one time step (these probabilities are not shown in Figure 25 for fear that they would clutter the picture). Typically, in a Discrete Time Markov Chain, these single-step transition probabilities are known. In our case, however, these single-step transition probabilities are unknown – they are the quantities that we want to estimate.

Estimating the single-step transition probabilities of our model was not a goal of this research; however, we nonetheless present an experimental design that would allow one to estimate these probabilities by using a suitable combat simulation. First, we note that the single-step transition probabilities that we seek to find will likely depend on three factors: the composition of the sensor package executing the search, the type of target for which the sensor package is searching, and the UTZ classification of the NAI being searched. Next, we propose that data from a high-resolution simulation be used in order to estimate the single-step transition probabilities for each sensor package/target type/UTZ classification combination.

Now, in order to estimate the single-step transition probabilities for one sensor package/target type/UTZ classification, one would use an approach similar to a Monte Carlo simulation; pseudo code for this process can be found in Figure 26. Specifically, one would build a scenario in a high-resolution simulation by selecting a sensor package, target type, and UTZ classification for the NAI. A large number of targets of the selected target type should initially be placed in an Undetected state. Prior to starting the simulation, a time step size, Δt, should be selected (it is not necessary, here, that Δt be a small number; large values of Δt may be useful, as well). Then, one would allow the simulation to run (allowing the sensor package to search for all targets) for many time steps. During the simulation, the state of each target at the end of each time step should be recorded. Thus, upon completion of the simulation, one would have data that shows, for each target, the acquisition state of the target after each time step. Table 2 shows an example of recorded data from a hypothetical simulation run to estimate single-step transition probabilities.

```
FOR EACH TERRAIN TYPE {
     FOR EACH SENSOR PACKAGE {
          FOR EACH TARGET TYPE (T) {
               INITIALIZE MANY TARGETS OF TYPE T
               PLACE ALL TARGETS IN UNDETECTED STATE
               SELECT A TIME STEP SIZE
               SELECT A SIMULATION RUN TIME (MANY TIME STEPS)
               BEGIN THE HIGH-RESOLUTION SIMULATION {
                    FOR EACH TIME STEP {
                         FOR EACH TARGET {
                              RECORD THE STATE
                         }
                    }
               } END THE HIGH-RESOLUTION SIMULATION
          }
     }
}
```

Figure 26. Pseudo Code for Estimating Single-Step Transition Probabilities

	Time Step 1	Time Step 2	Time Step 3	Time Step 4	Time Step 5
Target 1	U	D	R	I	U
Target 2	U	U	I	U	D
Target 3	U	D	U	D	U
Target 4	U	R	I	I	I
Target 5	U	I	I	U	D
Target 6	U	U	D	D	D
Target 7	U	U	U	U	U

U = Undetected D = Detected R = Recognized I = Identified

Table 2. Data From a Hypothetical Simulation Run to Estimate Single-Step Transition Probabilities

If we let t be the number of targets and n be the number of time steps, then the data in Table 2 essentially represents $t \cdot n$ state transitions. For each of these state transitions, we know the beginning state and the ending state. Thus, in order to calculate the single-step transition probabilities from data such as that in Table 2, we first need to sort all of these state transitions by the beginning state. For our simple model, we would have four sets of state transitions: those beginning in the Undetected, Detected, Recognized, and Identified states, respectively. Then, for the transitions beginning in the Undetected state, we can further sort by the ending state and count the number of transitions that end at the Undetected, Detected, Recognized, and Identified states, respectively. Each of these four numbers can then be divided by the total number of transitions beginning in the Undetected state to estimate each of the single-step transition

probabilities for transitions emanating from the Undetected state. Figure 27 illustrates a sample calculation of this type.

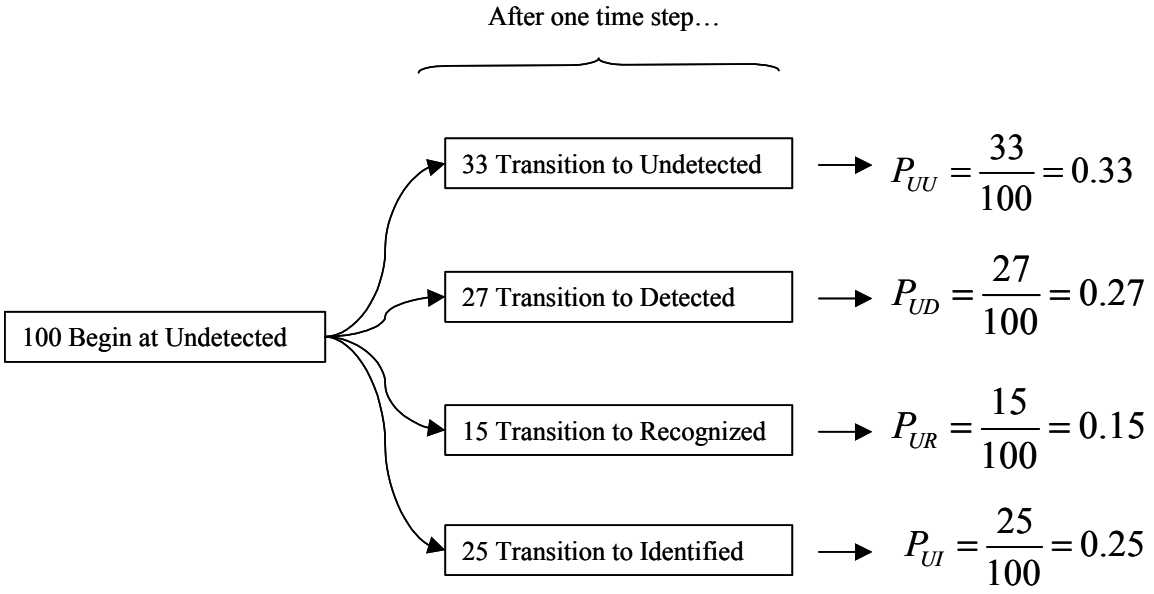


Figure 27. Sample Calculation of Single-Step Transition Probabilities

We can repeat these calculations for transitions beginning in all other states in order to estimate all of the single-step transition probabilities of our model. This entire process could then be repeated for a different sensor package/target type/UTZ classification. Once the single-step transition probabilities have been calculated, one can use the analytical results on Discrete Time Markov Chains presented in (Ross, 2003) to answer Questions #1 and #3 asked of this model.

e. Model Development – The Event Step Model

Our Event Step Model is similar to a Discrete Time Markov Chain, but each arc has a single-step transition time associated with it instead of a single-step transition probability. Thus, in order to develop our Event Step Model, we retain the four acquisition states, but reformat the arcs in our picture; this Event Step Model is illustrated in Figure 28. In our Discrete Time Markov Chain Model, the target can "jump" to any

state in a single time step. However, when modeling this process using an Event Step model, we assume that the target must proceed sequentially through the acquisition states (from Undetected to Detected to Recognized to Identified) – the target is not permitted to "skip" an acquisition state. The reason me make this assumption stems purely from the fact that, as mentioned earlier, the definitions of (JP 1-02, 2001) suggest a "hierarchy" of acquisition states that should be followed. However, instead of moving successively to the next "higher" acquisition state, the target could become Undetected, effectively starting the problem over again. Typically, in an Event Step Model, the transition times between states are known. In our case, however, these transition times are unknown – they are the quantities that we want to estimate.

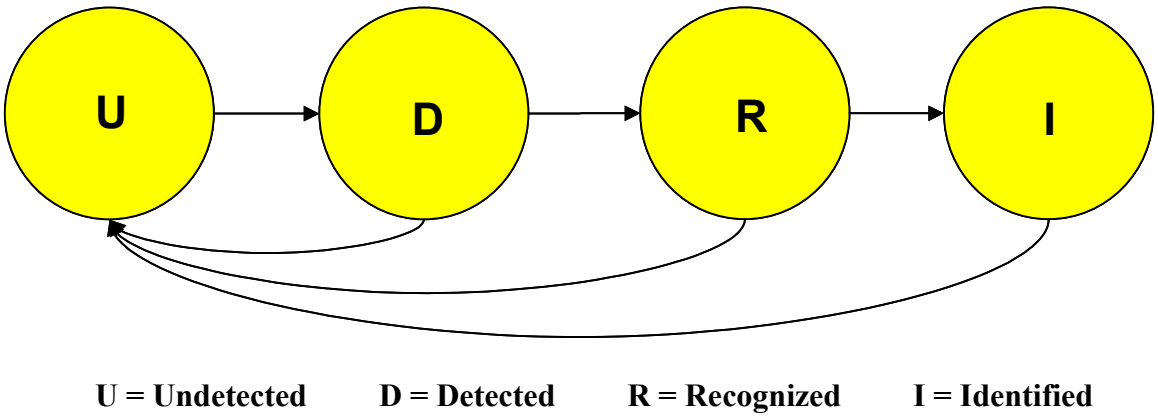


Figure 28. Event Step Model of Target Acquisition States

Just as with our Discrete Time Markov Chain Model, estimating the single-step transition times was not a goal of this research; however, we nonetheless present an experimental design that would allow one to estimate these probabilities by using a suitable combat simulation. As with the Discrete Time Markov Chain Model, we assume that the single-step transition times that we seek to find will be different for each sensor package/target type/UTZ classification combination, and we propose that data from a high-resolution simulation be used in order to estimate the single-step transition times for each of these combinations.

Now, in order to estimate the single-step transition times for one sensor package/target type/UTZ classification, one would again use an approach similar to a Monte Carlo simulation; pseudo code for this process can be found in Figure 29. Specifically, one would build a scenario in a high-resolution simulation by selecting a sensor package, target type, and UTZ classification for the NAI. A large number of targets of the selected target type should initially be placed in an Undetected state. Then, one would allow the simulation to run (allowing the sensor package to search for all targets) for a sufficient length of time. Whenever any target undergoes a state change during the simulation, the new state of the target should be recorded as well as the current simulation time. Thus, upon the termination of the simulation, one would have, for each target, a list of the states visited and the time spent in each state. Table 3 shows an example of recorded data from a hypothetical simulation run to estimate single-step transition times.

At the end of the simulation run, one can then calculate statistics on the transition times (such as the mean and variance) recorded for each ending acquisition state. This will give an estimate as to the expected transition time to each state as well as insight as to the *distribution* of transition times to each state. This entire process could then be repeated for a different sensor package/target type/UTZ classification. Once these single-step transition times (and distributions of single-step transition times) have been calculated, one can use simple calculations to answer Questions #2 and #3 of this model.

```
FOR EACH TERRAIN TYPE {

     FOR EACH SENSOR PACKAGE {

          FOR EACH TARGET TYPE (T) {

               INITIALIZE MANY TARGETS OF TYPE T
               PLACE ALL TARGETS IN UNDETECTED STATE
               SELECT A SIMULATION RUN TIME

               BEGIN THE HIGH-RESOLUTION SIMULATION {

                    FOR EACH STATE TRANSITION {

                         RECORD THE NEW STATE OF THE TARGET
                         RECORD THE SIMULATION TIME

                    }

               } END THE HIGH-RESOLUTION SIMULATION

          }

     }

}
```

Figure 29. Pseudo Code for Estimating Single-Step Transition Times

	Transition 1 / Simulation Time	Transition 2 / Simulation Time	Transition 3 / Simulation Time	Transition 4 / Simulation Time	Transition 5 / Simulation Time
Target 1	D / 4.5	U / 5.7	D / 17.4	R / 17.4	I / 17.4
Target 2	D / 0.5	U / 1.5	D / 3.6	U / 6.6	D / 6.8
Target 3	D / 10.5	R / 14.0	I / 14.5	U / 39.9	D / 44.6
Target 4	D / 8.7	R / 8.7	U / 8.9	D / 39.0	R / 41.4

U = Undetected D = Detected R = Recognized I = Identified

Table 3. Data From a Hypothetical Simulation Run to Estimate Single-Step Transition Times

3. Aggregate Flow Model

a. Problem Description

A large number of sensors are arrayed against a large number of targets in a particular AO. The blue force knows the number and type of sensors it has available, but does not know the number and type of targets that exist in the AO.

b. Model Objectives

Our model is designed to answer the following questions:

1. How many targets are detected, recognized, or identified at any given time?

2. What is the rate of change in the number of targets detected, recognized, or identified at any given time?

3. How is this rate of change dependent upon the number of targets in each state at a given time?

c. Approach

Our approach to answering the three questions above will be significantly different than that of our first two models. For this model, we will model the "flow" of targets between acquisition states using a system of differential equations. In essence, we will be seeking the urban target acquisition equivalent to Lanchester's equations for attrition modeling. Lanchester's equations propose that the rate at which a force is depleted is proportional to the size of the enemy force and the individual capability of the enemy. More detailed information on Lanchester's equations can be found in (Taylor, 1983). In our model, we will develop equations that relate the rate at which targets enter a given acquisition state to the number of targets in each acquisition state.

d. Model Development

As in our previous models, we consider four acquisition states for targets: Undetected, Detected, Recognized, and Identified. However, we realize that there is the possibility that a target may not be in any of these states; specifically, once it has been killed. Thus, in order to account for all targets, we can add a fifth acquisition state: Killed (alternatively, we could allow for targets "leaving" the system). Next, we represent the "flow" of targets from one acquisition state to another using arcs to connect the acquisition states; this representation is shown in Figure 30.

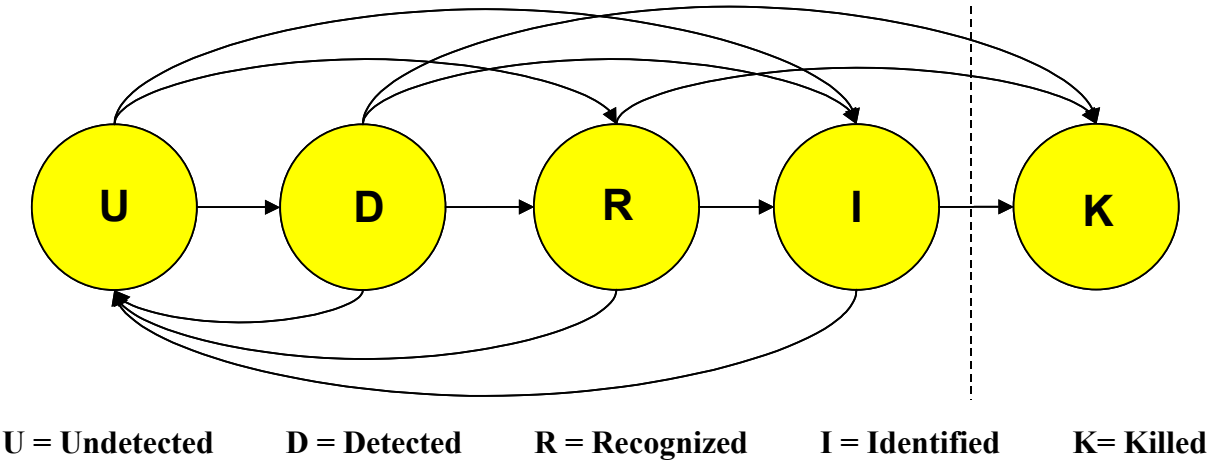


Figure 30. A Representation of the Flow of Targets Between Acquisition States

At this point, a few clarifications of Figure 30 are required. First, in this model, a target is allowed to "skip" acquisition states (for example, a target can jump from the Undetected state to the Recognized state, effectively skipping the Detected state). We adopt this convention because we will be considering time steps in this model (as in the Discrete Time Markov Chain Model), not event steps (as in the Event Step Model). Additionally, we note that a target cannot leave the Killed state – this state effectively allows us to keep the total number of targets in the system constant at any given time.

Let x_i be the number of targets in state i at a given time. Then $\frac{dx_i}{dt}$ is the instantaneous rate of change of the number of targets in state i with respect to time. Now, we consider the "balance of flow" at each acquisition state. The rate of change of the number of targets in state i for a given time period is equal to the number of targets entering state i in that time period *minus* the number of targets leaving state i in that time period. It is prudent to assume that the number of targets entering and leaving state i in a given time period would be dependent upon the number of targets in *each state* at the beginning of the time period.

We further assume in the development of this model, that the number of targets entering and leaving each state in a given time period is *linearly* dependent on the number of targets in each state at the beginning of the time period. While this assumption may seem to be impractical, it should be noted that Lanchester's basic equations for attrition modeling assume linear relationships, and Lanchester's models have certainly proven over time to be a good predictor of attrition.

Thus, we conjecture that the instantaneous rate of change of the number of targets in state i at time t is linearly dependent upon the number of targets in each state at time t. This conjecture allows us to formulate a system of differential equations modeling the flow of targets. For our particular model, we have five differential equations (one for each acquisition state):

$$\frac{dx_U}{dt} = a_{UU}x_U + a_{UD}x_D + a_{UR}x_R + a_{UI}x_I + a_{UK}x_K$$
$$\frac{dx_D}{dt} = a_{DU}x_U + a_{DD}x_D + a_{DR}x_R + a_{DI}x_I + a_{DK}x_K$$
$$\frac{dx_R}{dt} = a_{RU}x_U + a_{RD}x_D + a_{RR}x_R + a_{RI}x_I + a_{RK}x_K$$
$$\frac{dx_I}{dt} = a_{IU}x_U + a_{ID}x_D + a_{IR}x_R + a_{II}x_I + a_{IK}x_K$$
$$\frac{dx_K}{dt} = a_{KU}x_U + a_{KD}x_D + a_{KR}x_R + a_{KI}x_I + a_{KK}x_K.$$

(The coefficients a_{ij} in the above equations are "acquisition coefficients" that simply describe the extent of the linear relationship between $\frac{dx_i}{dt}$ and x_j; they are analogous to the attrition coefficients of Lanchester's equations.)

A similar system of differential equations could be developed for a model that includes more or less acquisition states – it would simply involve more or less equations. Because we have assumed linear relationships, this system of differential equations can be solved analytically to derive an equation for each x_i as a function of t. Using these analytical results, we can predict the number of targets in each acquisition state at a given time as well as the instantaneous rate at which targets enter or leave each state at a given time.

In order to develop our model to completion, however, extensive work would need to be done to estimate the acquisition coefficients in the system of differential equations. While we did not pursue the estimation of these acquisition coefficients as a part of this research, we still propose a method by which these coefficients can be estimated. First, we note that the acquisition coefficients that we seek to find will not be constant. Instead, they will likely depend on four factors: the composition of the set of sensors executing the search, the composition of the target set for which the set of sensors is searching, the UTZ classification of the AO being searched, and the size of the AO being searched. Next, just as with our previous models, we propose that data from a high-resolution simulation be used in order to estimate the acquisition coefficients for each sensor set/target set/UTZ classification/AO size combination; pseudo code for this process can be found in Figure 31.

To estimate the acquisition coefficients for one combination of sensor set, target set, UTZ classification, and AO size, one would first select an urban template representative of a particular UTZ classification and fix its size. Next, the composition of the set of sensors and the set of targets should be fixed. All targets should be placed in the Undetected state at the onset of the simulation. Prior to starting the simulation, a sufficiently small time step size, Δt, should be selected. Then, the simulation should be permitted to run for an indefinite period of time. After each Δt time units, one should

record the time, t, and the number of targets in state i at time t, $x_i(t)$, for each state i. Table 4 shows an example of the data recorded for a hypothetical simulation run with $\Delta t = 1$.

```
FOR EACH TERRAIN TYPE {
     FOR EACH SENSOR SET {
          FOR EACH TARGET SET {
               PLACE ALL TARGETS IN UNDETECTED STATE
               SELECT A TIME STEP SIZE
               SELECT A SIMULATION RUN TIME (MANY TIME STEPS)
               BEGIN THE HIGH-RESOLUTION SIMULATION {
                    FOR EACH TIME STEP {
                         RECORD THE SIMULATION TIME
                         FOR EACH ACQUISITION STATE {
                              RECORD THE NUMBER OF TARGETS
                                   IN THAT STATE
                         }
                    }
               } END THE HIGH-RESOLUTION SIMULATION
          }
     }
}
```

Figure 31. Pseudo Code for Estimating Acquisition Coefficients

	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$
$x_U(t)$	100	95	92	87	88
$x_D(t)$	0	3	3	7	6
$x_R(t)$	0	1	2	3	2
$x_I(t)$	0	1	2	2	2
$x_K(t)$	0	0	1	1	2

Table 4. Data From a Hypothetical Simulation Run to Estimate Flow Rates with $\Delta t = 1$

From the data in Table 4, one can then estimate $\frac{dx_i}{dt}$ at any time using an approximation of the derivative:

$$\frac{dx_i}{dt} \simeq \frac{x_i(t+\Delta t) - x_i(t)}{\Delta t}.$$

If we have selected Δt small enough, then this approximation will be close to the true value of the derivative. Then, for each state i, and for any time, one can conduct a linear regression using $\frac{dx_i}{dt}$ as the dependent variable and x_U, x_D, x_R, x_I, x_K as the independent variables. The coefficients resulting from the linear regression are the estimated acquisition coefficients in the original system of differential equations.

The entire process described above for estimating acquisition coefficients can be repeated for each sensor set/target set/UTZ classification/AO size combination.

Once the acquisition coefficients have been estimated, the system of differential equations in this model (and the subsequent derived functions for each x_i as a function of t) can be used to answer the three questions of this model.

IV. ACCOMPLISHMENTS AND RECOMMENDED FURTHER RESEARCH

A. ACCOMPLISHMENTS

1. Probability of Line of Sight in Urban Terrain

Through the development and validation of the UPPS, we accomplished our objective of demonstrating that the PLOS methodology developed and used for open terrain could be extended to produce useful PLOS estimates for urban terrain. The purpose of this paper was not to perform a thorough analysis of urban PLOS estimates. Rather, our intent was simply to demonstrate the usefulness of our product, the UPPS, and present some interesting results unique to urban PLOS that should be addressed in future research. Following is a summary of our accomplishments in our research of urban PLOS:

1. We developed the UPPS, demonstrating that the PLOS methodology developed for open terrain could be extended to produce PLOS estimates for urban terrain.

2. We produced PLOS curves for the three available urban templates representative of different urban terrain types.

3. We noted how the urban PLOS curves we generated could likely be approximated well with a simple, analytical, closed-form function.

4. By producing urban PLOS estimates using an existing combat simulation, Combat XXI, we conducted a "sanity check" to ensure that the UPPS urban PLOS estimates were accurate. This comparison can also be considered a first step in the validation of the UPPS.

5. We modified the UPPS to accommodate elevated (specifically, rooftop) targets and described how urban PLOS curves could be generated for sensor elevation-target elevation combinations.

6. With the help of TerraSim, we "mounted" urban templates onto underlying terrain and suggested further testing using the UPPS in order to assess how the underlying terrain skin affects urban PLOS.

7. We derived analytical functions that use urban PLOS estimates to approximate the time until LOS is gained and lost for a given sensor elevation and urban terrain type.

2. Stochastic and Analytical Models for Target Acquisition in Urban Terrain

By developing the stochastic and analytical models for urban target acquisition that are presented in this paper, we met our objective of identifying models that had the potential to decrease simulation run time for large-scale urban scenarios and/or could lend insight into target acquisition in urban terrain. When developing our stochastic and analytical models for urban target acquisition, it was not our intent to develop our models to completion. Instead, we simply carried out a superficial exploration and presented in this paper those models that, in our opinion, showed promise as a useful simulation model or stand-alone decision tool. Specifically, we developed the following stochastic and analytical models for representing urban target acquisition:

1. A cumulative distribution function model that can estimate the proficiency of a given set of sensors against a given set of targets in a particular urban area.

2. Entity-level state transition models that can represent the movement of a target among acquisition states.

3. An aggregate flow model that may be used to assess the flow of targets from one acquisition state to another.

B. RECOMMENDED FURTHER RESEARCH

1. Probability of Line of Sight in Urban Terrain

Again, the purpose of this paper was not to carry out an extensive analysis of urban PLOS estimates. Instead, we focused on presenting an "on the surface"

preliminary analysis of topics relevant to urban PLOS. Following are urban PLOS topics to which further research should be dedicated:

1. A thorough validation (potentially using a production version of Combat XXI) should be conducted to examine the accuracy of the UPPS. Additionally, an analysis of the differences between urban PLOS estimates generated by the UPPS and Combat XXI should be completed.

2. Extensive "grunt work" needs to be done to generate all appropriate PLOS curves for each sensor elevation-target elevation combination for the three existing urban templates: City Core, Commercial Ribbon, and Outlying High-Rise Area.

3. A study should be done to assess how the distribution of observations into range bins (shown in Figure 10) affects urban PLOS estimates. Additionally, one could pursue whether there is a more efficient way to generate sensor-target pairs that would result in a more uniform distribution than the "normal-like" distribution seen in Figure 10.

4. A detailed investigation into the "strange behavior" of some of the urban PLOS curves should be completed. Specifically, analysis should be done to determine why PLOS curves for lower sensor elevations sometimes yield a higher PLOS than for higher sensor elevations and why, for some curves, PLOS actually increases with range.

5. Curve fitting should be carried out on the urban PLOS curves for the available urban templates representative of each UTZ classification in an attempt to approximate urban PLOS estimates with a closed-form function. An important subproblem is how to quantify the urban terrain type when developing these closed-form functions. One possible solution is to use statistics from each urban template as parameters (e.g. mean building height, coverage factor, etc.). Analysis should be done on the errors introduced by a function approximation to determine if the error is within an acceptable tolerance.

6. A thorough analysis should be accomplished to determine if the underlying terrain skin in an urban area makes a significant difference in the estimates of urban PLOS.

7. A validation should be conducted of the analytical estimates of the time until LOS gained or lost suggested in this research.

2. Stochastic and Analytical Models for Target Acquisition in Urban Terrain

As our work in developing our stochastic and analytical urban target acquisition models was largely exploratory in nature, there is obviously a plethora of additional research and "brainstorming" to which further research could be dedicated. We present a few topics of additional research that apply specifically to the models developed in this paper:

1. Our cumulative distribution function model should be further developed to answer the following question: "Given that a set of sensors has detected x targets, how many targets *actually exist* in the AO?" If our cumulative distribution function model is extended to answer this question, it could prove to be an extremely useful tool for the Future Force commander.

2. Testing should be accomplished to estimate the single-step transition probabilities and single-step transition times proposed by our entity-level state transition models for a myriad of sensor packages, target types, and urban terrain types.

3. Testing should be accomplished to estimate the acquisition coefficients suggested by our aggregate flow model for different sensor sets, target sets, and urban terrain types.

APPENDIX A. USER'S MANUAL FOR THE UPPS

A. UPPS INPUT

The input to the UPPS is a text file of the format shown in Figure 32. The first line of the file has three numbers: the x-dimension of the urban template, the y-dimension of the urban template, and the number of buildings in the urban template. Following is a line to describe the location and height of each building (in the case of the urban template described by the text file in Figure 32, we have 209 additional lines). The first number in each of these lines is the number of corners of the building. Subsequently, the x- and y-coordinates of each corner of the building are listed (in a clockwise or counterclockwise pattern). The final number on each line is the height of the building. For example, the first building listed in the text file in Figure 32 has four corners at coordinates (1,38), (1,1), (47,1), and (47,38), respectively, and has a height of 29.2608 meters.

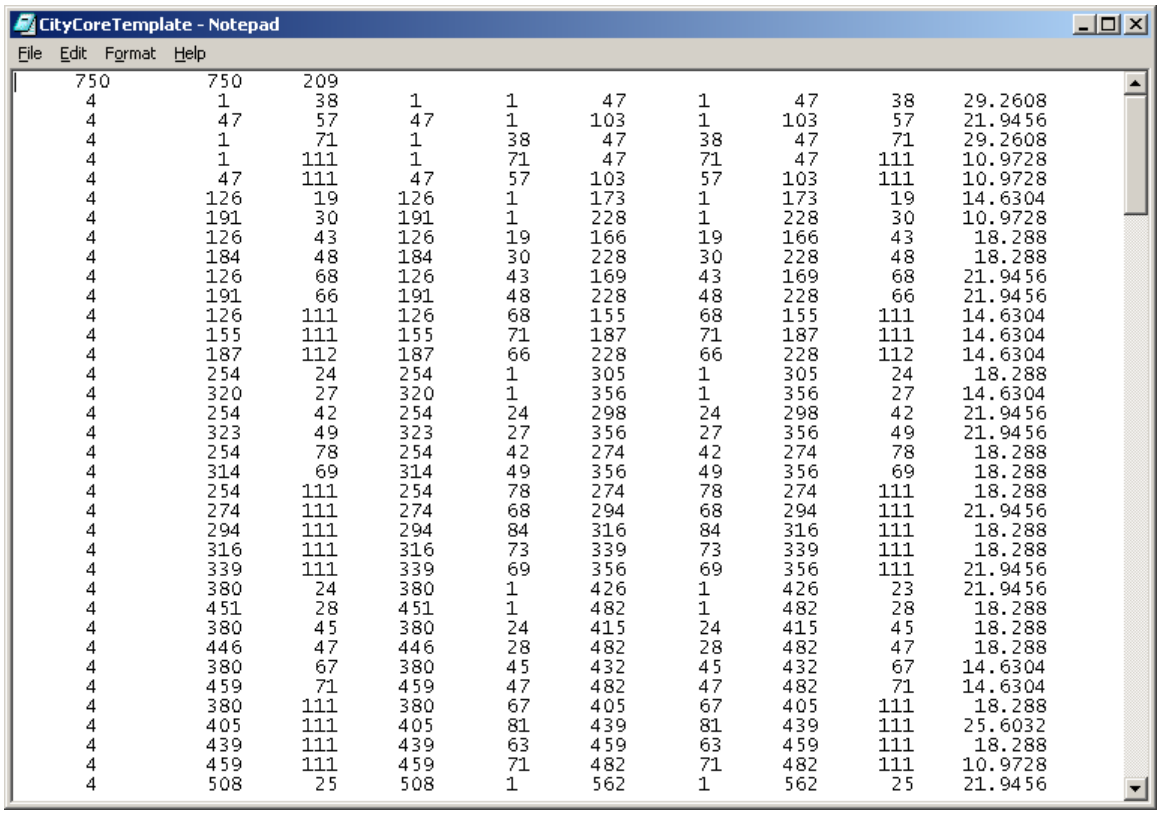


Figure 32. UPPS Input File for the City Core Urban Template

B. UPPS OUTPUT

Figure 33 shows example output from the UPPS. The output file contains urban PLOS estimates for one particular urban template/sensor elevation/target elevation combination. The output file contains three columns. The first column lists the range bin, the second column lists the PLOS estimate for that range bin, and the third column lists the number of observations in that particular range bin. For example, the first line of this output file tells us that in the 0 – 5-meter range bin there were 1470 observations with a PLOS estimate of 0.9782. The second line of this output file tells us that in the 5 – 10-meter range bin there were 4166 observations with a PLOS estimate of 0.9618.

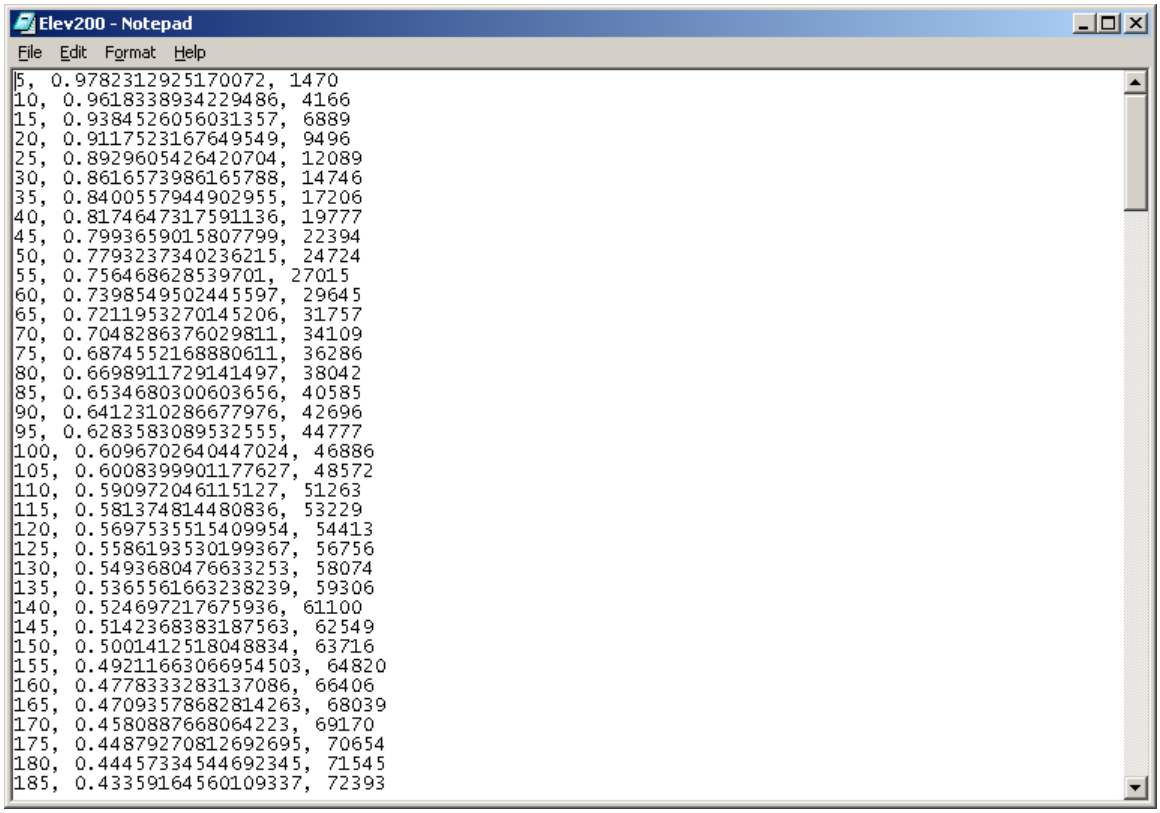


Figure 33. UPPS Output File for the City Core Urban Template, Sensor Elevation 200 meters, Target Elevation 0 meters

C. JAVA CLASSES

The Java classes used to develop the UPPS were of three different types: existing Java classes, existing Simkit classes, and new classes developed "from scratch" specifically for this simulation.

1. Existing Java Classes

Following is a list of the existing Java classes used for the UPPS; javadocs for these classes can be found at http://java.sun.com/j2se/1.4.2/docs/api/index.html.

For reading in the data:

FileReader

BufferedReader

StringTokenizer

Integer

Double

For Exception handling:

FileNotFoundException

CloneNotSupportedException

RuntimeException

For basic mathematical operations:

Math

For geometry purposes:

Polygon

Line2D

2. Existing Simkit Classes

Simkit is a Java package designed for use in discrete event simulations (more information on Simkit can be found at http://diana.gl.nps.navy.mil/Simkit/). For the purposes of the UPPS, Simkit was used for random number generation and calculating statistics. Following is a list of the existing Simkit classes used in the UPPS; javadocs for these classes can be found at http://diana.gl.nps.navy.mil/Simkit/doc/.

For generating random numbers:

RandomVariate

RandomVariateFactory

For calculating statistics:

SimpleStatsTally

3. New Classes

Following is a summary of the new classes built for the UPPS. Java code for all of these classes is included in Appendix B.

a. Point3D

Point3D objects represent points (or vectors) in 3-dimensional space.

b. Entity

Entity objects represent entities on the battlefield. This class is a super class of the Sensor and Target classes, so that instance variables and methods common to these two classes would not have to be written twice. This class contains two different instance methods for computing whether LOS exists between two entities (hasLineOfSightTo and hasLOSTo)

c. Sensor

Sensor objects represent sensors on the battlefield. This is a simple class, but it can be modified if we want to construct more "complex" sensors.

d. Target

Target objects represent targets on the battlefield. Again, this is a simple class, but it can be modified if we want to construct more "complex" targets.

e. Building

Building objects represent buildings in an urban template. Each building has an array of line segments (Line2D objects) representing its walls (as seen from a top view) and a Polygon representing its "footprint" (the shape of the building as seen from a top view). These two instance variables are redundant, but the redundancy makes the code of the UPPS easier to develop and understand.

f. PLOS

This is the pure execution class that executes the simulation. This is the class that a user will modify to change the sensor elevation, target elevation, range bin size, and other attributes of the UPPS.

4. LOS Algorithms

Following is a more detailed description of the two LOS algorithms used in the UPPS. Again, the Java code for these methods can be found in the Entity class in Appendix B.

a. hasLineOfSightTo

This LOS method should be used when incorporating underlying terrain skin into LOS calculations. The LOS algorithm in this method essentially "steps" along the line from sensor to target and checks the slope at each step (if any of these

intermediate slopes is greater than the slope from sensor to target, then LOS does not exist). The "steps" taken by this algorithm are at every gridline crossed.

b. hasLOSTo

This LOS method should be used when assuming that the underlying terrain skin will have a negligible effect on LOS. The LOS algorithm in this method takes the same approach as the hasLineOfSightTo method ("stepping" along the line from sensor to target and checking slopes). However, the only points checked by this method along the line from sensor to target are where the sensor-target line intersects a building wall. This method runs *much* faster than the hasLineOfSightTo method.

APPENDIX B. JAVA CODE FOR THE UPPS

A. CLASS POINT3D

```
/*
 * Point3D.java
 *
 * Created on October 18, 2003, 5:50 PM
 */

public class Point3D implements Cloneable {

  private double x;
  private double y;
  private double z;

  public Point3D() {
    this(0.0, 0.0);
  }

  public Point3D(double xCoord, double yCoord) {
    this(xCoord, yCoord, 0.0);
  }

  public Point3D(double xCoord, double yCoord, double zCoord) {
    setPoint(xCoord, yCoord, zCoord);
```

```
}

public Point3D(Point3D point) {
  this(point.getX(), point.getY(), point.getZ());
}

public Point3D setPoint(double xCoord, double yCoord, double zCoord) {
  setX(xCoord);
  setY(yCoord);
  setZ(zCoord);
  return this;
}

public Point3D setPoint(Point3D point) {
  return setPoint(point.getX(), point.getY(), point.getZ());
}

public double getX() {
  return x;
}

public void setX(double newX) {
  x = newX;
}
```

```
public double getY() {
   return y;
}

public void setY(double newY) {
   y = newY;
}

public double getZ() {
   return z;
}

public void setZ(double newZ) {
   z = newZ;
}

public Point3D incrementBy(double deltaX, double deltaY, double deltaZ) {
   x += deltaX;
   y += deltaY;
   z += deltaZ;
   return this;
}
```

```
public Point3D incrementBy(Point3D delta) {
  return incrementBy(delta.getX(), delta.getY(), delta.getZ());
}

public Point3D decrementBy(double deltaX, double deltaY, double deltaZ) {
  x -= deltaX;
  y -= deltaY;
  z -= deltaZ;
  return this;
}

public Point3D decrementBy(Point3D delta) {
  return decrementBy(delta.getX(), delta.getY(), delta.getZ());
}

public Point3D scalarMultiply(double alpha) {
  x *= alpha;
  y *= alpha;
  z *= alpha;
  return this;
}

public double distanceFrom(double xCoord, double yCoord, double zCoord) {
  return Math.sqrt((xCoord - x)*(xCoord - x) + (yCoord - y)*(yCoord - y)
```

```
                + (zCoord - z)*(zCoord - z));
}

public double distanceFrom(Point3D point) {
  return distanceFrom(point.getX(), point.getY(), point.getZ());
}

public double XYDistanceFrom(double xCoord, double yCoord) {
  return Math.sqrt((xCoord - x)*(xCoord - x) + (yCoord - y)*(yCoord - y));
}

public double XYDistanceFrom(Point3D point) {
  return XYDistanceFrom(point.getX(), point.getY());
}

public double slopeTo(double xCoord, double yCoord, double zCoord) {
  return (zCoord - z)/XYDistanceFrom(xCoord, yCoord);
}

public double slopeTo(Point3D point) {
  return slopeTo(point.getX(), point.getY(), point.getZ());
}

public double XYSlopeTo(double xCoord, double yCoord) {
```

```
    return (yCoord - y)/(xCoord - x);
  }

  public double XYSlopeTo(Point3D point) {
    return XYSlopeTo(point.getX(), point.getY());
  }

  public Object clone() {
    Point3D answer;
    try {
      answer = (Point3D)super.clone();
    }
    catch(CloneNotSupportedException e) {
      throw new RuntimeException("This class does not implement cloneable");
    }
    return answer;
  }

  public String toString() {
    return "Point3D:  (" + x + ", " + y + ", " + z + ")";
  }


}
```

B. CLASS ENTITY

```
/*
 * Entity.java
 *
 * Created on October 18, 2003, 3:03 AM
 */

import java.awt.geom.*;

public class Entity {

  private Point3D location;

  private static double getXIntersectPoint(Line2D a, Line2D b) {
    if (a.getX1() == a.getX2()) {
      return a.getX1();
    }
    if (b.getX1() == b.getX2()) {
      return b.getX1();
    }
    double slopeA, slopeB;
    slopeA = (a.getY2() - a.getY1())/(a.getX2() - a.getX1());
    slopeB = (b.getY2() - b.getY1())/(b.getX2() - b.getX1());
    return (slopeA*a.getX1() - slopeB*b.getX1() + b.getY1() - a.getY1())/(slopeA -
slopeB);
```

```
  }

  private static double getYIntersectPoint(Line2D a, Line2D b) {
    if (a.getX1() == a.getX2()) {
      return ((b.getY2() - b.getY1())/(b.getX2() - b.getX1()))*(getXIntersectPoint(a, b) -
b.getX1()) + b.getY1();
    }
    else {
      return ((a.getY2() - a.getY1())/(a.getX2() - a.getX1()))*(getXIntersectPoint(a, b) -
a.getX1()) + a.getY1();
    }
  }

  public Entity(Point3D initialLocation) {
    setLocation(initialLocation);
  }

  public Point3D getLocation() {
    return (Point3D)location.clone();
  }

  public void setLocation(Point3D newLocation) {
    location = (Point3D)newLocation.clone();
  }
```

```
  public boolean hasLineOfSightTo(Entity target, double[][] elevation) {
    double XYSlopeToTarget = location.XYSlopeTo(target.getLocation());
    Point3D intermediateLocation = new Point3D(location);
    double targetX = target.getLocation().getX();
    double targetY = target.getLocation().getY();
    double sensorX = location.getX();
    double sensorY = location.getY();
    double nextXLine;
    double nextYLine;


    //Sensor Upper-Left, Target Lower-Right
    if (targetX > sensorX && targetY < sensorY) {
      nextXLine = Math.ceil(sensorX);
      nextYLine = Math.floor(sensorY);
      while (nextXLine < targetX || nextYLine > targetY) {

        if (XYSlopeToTarget >= intermediateLocation.XYSlopeTo(nextXLine,
nextYLine)) {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
            (nextXLine - intermediateLocation.getX())/(targetX -
intermediateLocation.getX())));
          nextXLine += 1.0;
```

```
        }
        else {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocation).scalarMultiply(
            (nextYLine - intermediateLocation.getY())/(targetY - intermediateLocation.getY())));
          nextYLine -= 1.0;
        }

        if (intermediateLocation.getZ() <

elevation[(int)Math.round(intermediateLocation.getX())][(int)Math.round(intermediateLocation.getY())]) {
          return false;
        }

      }
      return true;
    }

    //Sensor Lower-Left, Target Upper-Right
    else if (targetX > sensorX && targetY > sensorY) {
      nextXLine = Math.ceil(sensorX);
      nextYLine = Math.ceil(sensorY);
```

```
        while (nextXLine < targetX || nextYLine < targetY) {

            if (XYSlopeToTarget < intermediateLocation.XYSlopeTo(nextXLine,
nextYLine)) {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
                (nextXLine - intermediateLocation.getX())/(targetX -
intermediateLocation.getX())));
                nextXLine += 1.0;
            }

            else {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
                (nextYLine - intermediateLocation.getY())/(targetY -
intermediateLocation.getY())));
                nextYLine += 1.0;
            }

            if (intermediateLocation.getZ() <

elevation[(int)Math.round(intermediateLocation.getX())][(int)Math.round(intermediateL
ocation.getY())]) {
                return false;
```

```
        }


      }
      return true;
    }

    //Sensor Right, Target Left
    else {
      return target.hasLineOfSightTo(this, elevation);
    }
  }

  public boolean hasLOSTo(Entity target, Building[] building) {
    Point3D intermediateLocation = new Point3D();
    Line2D.Double sensorTargetLine = new Line2D.Double(
      location.getX(), location.getY(), target.getLocation().getX(),
target.getLocation().getY());
    double intersectPoint;

    for(int i = 0; i < building.length; i++) {
      for(int j = 0; j < building[i].getWall().length; j++) {
        intermediateLocation.setPoint(location);

        if (sensorTargetLine.intersectsLine(building[i].getWall()[j])) {
```

```
            if (sensorTargetLine.getX1() != sensorTargetLine.getX2()) {
              intersectPoint = getXIntersectPoint(sensorTargetLine,
building[i].getWall()[j]);

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
                (intersectPoint - intermediateLocation.getX())/(target.getLocation().getX() -
intermediateLocation.getX())));
            }
            else {
              intersectPoint = getYIntersectPoint(sensorTargetLine,
building[i].getWall()[j]);

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
                (intersectPoint - intermediateLocation.getY())/(target.getLocation().getY() -
intermediateLocation.getY())));
            }

            if (intermediateLocation.getZ() < building[i].getHeight()) {
                return false;
            }
          }
        }
      }
```

```
    return true;

}

public boolean hasLineOfSightTo(Entity target, Building[] building) {
    double XYSlopeToTarget = location.XYSlopeTo(target.getLocation());
    Point3D intermediateLocation = new Point3D(location);
    double targetX = target.getLocation().getX();
    double targetY = target.getLocation().getY();
    double sensorX = location.getX();
    double sensorY = location.getY();
    double nextXLine;
    double nextYLine;
    boolean inBuilding;
    int i;

    //Sensor Upper-Left, Target Lower-Right
    if (targetX > sensorX && targetY < sensorY) {
      nextXLine = Math.ceil(sensorX);
      nextYLine = Math.floor(sensorY);
      while (nextXLine < targetX || nextYLine > targetY) {
        inBuilding = false;
        i = 1;
```

```
        if (XYSlopeToTarget >= intermediateLocation.XYSlopeTo(nextXLine,
nextYLine)) {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
          (nextXLine - intermediateLocation.getX())/(targetX -
intermediateLocation.getX())));
          //System.out.println("Hit Y Line");
          nextXLine += 1.0;
        }
        else {
          //System.out.println("Hit X Line");

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
          (nextYLine - intermediateLocation.getY())/(targetY -
intermediateLocation.getY())));
          //System.out.println(intermediateLocation);
          nextYLine -= 1.0;
        }
        while (!inBuilding && i < building.length) {
          //System.out.println("Checking Building " + i);
          //System.out.println(intermediateLocation);
          if (building[i].contains(intermediateLocation)) {
            inBuilding = true;
            //System.out.println("found");
```

```
            if (intermediateLocation.getZ() < building[i].getHeight()) {
              return false;
            }
          }
          i++;
        }
      }
      return true;
    }

    //Sensor Lower-Left, Target Upper-Right
    else if (targetX > sensorX && targetY > sensorY) {
      nextXLine = Math.ceil(sensorX);
      nextYLine = Math.ceil(sensorY);
      while (nextXLine < targetX || nextYLine < targetY) {
        //System.out.println("Yo");
        inBuilding = false;
        i = 1;
        if (XYSlopeToTarget < intermediateLocation.XYSlopeTo(nextXLine,
nextYLine)) {

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocati
on).scalarMultiply(
            (nextXLine - intermediateLocation.getX())/(targetX -
intermediateLocation.getX())));
```

```
            //System.out.println("Hit Y Line");
            nextXLine += 1.0;
          }

          else {
            //System.out.println("Hit X Line");

intermediateLocation.incrementBy(target.getLocation().decrementBy(intermediateLocation).scalarMultiply(
            (nextYLine - intermediateLocation.getY())/(targetY - intermediateLocation.getY())));
            //System.out.println(intermediateLocation);
            nextYLine += 1.0;
          }

          while (!inBuilding && i < building.length) {
            //System.out.println("Checking Building " + i);
            //System.out.println(intermediateLocation);
            if (building[i].contains(intermediateLocation)) {
              inBuilding = true;
              //System.out.println("found");
              if (intermediateLocation.getZ() < building[i].getHeight()) {
                return false;
              }
            }
```

```
            i++;
          }
        }
        return true;
      }

      //Sensor Right, Target Left
      else {
        return target.hasLineOfSightTo(this, building);
      }
    }


    public String toString() {
      return location.toString();
    }

}
```

C. CLASS SENSOR

```
/*
 * Sensor.java
 *
 * Created on October 18, 2003, 2:40 AM
 */

public class Sensor extends Entity {

  private double maxRange;

  public Sensor() {
    this(new Point3D());
  }

  public Sensor(Point3D initialLocation) {
    this(initialLocation, Double.MAX_VALUE);
  }

  public Sensor(Point3D initialLocation, double initialMaxRange) {
    super(initialLocation);
    setMaxRange(initialMaxRange);
  }

  public double getMaxRange() {
```

```
    return maxRange;
  }

  public void setMaxRange(double newMaxRange) {
    maxRange = newMaxRange;
  }

  public String toString() {
    return super.toString() + "  Max Range:  " + maxRange;
  }
}
```

D. CLASS TARGET

```
/*
 * Target.java
 *
 * Created on October 18, 2003, 2:58 AM
 */

import java.awt.geom.*;

public class Target extends Entity {

  public Target() {
    this(new Point3D());
  }

  public Target(Point3D initialLocation) {
    super(initialLocation);
  }

  public String toString() {
    return super.toString();
  }

}
```

E. CLASS BUILDING

```
/*
 * Building.java
 *
 * Author:  Joe Mlakar
 *
 * Created on October 18, 2003, 2:29 AM
 */

import java.awt.*;
import java.awt.geom.*;

public class Building {

  private Polygon footprint;
  private Line2D[] wall;
  private double height;

  public Building(Polygon initialFootprint, Line2D[] initialWall, double initialHeight) {
    setFootprint(initialFootprint);
    setWall(initialWall);
    setHeight(initialHeight);
  }

  public Polygon getFootprint() {
```

```
    return footprint;
}

public void setFootprint(Polygon newFootprint) {
    footprint = newFootprint;
}

public Line2D[] getWall() {
    return (Line2D[])wall.clone();
}

public void setWall(Line2D[] newWall) {
    wall = (Line2D[])newWall.clone();
}

public double getHeight() {
    return height;
}

public void setHeight(double newHeight) {
    height = newHeight;
}

public boolean contains(double x, double y, double z) {
```

```
    return footprint.contains(x, y) && z <= height;
  }

  public boolean contains(Point3D point) {
    return contains(point.getX(), point.getY(), point.getZ());
  }
  /*
  public boolean contains(double x, double y) {
    Double buildingRightSide = new Double(footprint.getX() + footprint.getWidth());
    Double buildingTopSide = new Double(footprint.getY() + footprint.getHeight());
    if (footprint.contains(x, y) ||
       (buildingRightSide.equals(new Double(x)) && x >= footprint.getY() && x <
buildingTopSide.doubleValue()) ||
       (buildingTopSide.equals(new Double(y))&& y >= footprint.getX() && y <
buildingRightSide.doubleValue())) {
      return true;
    }
    else {
      return false;
    }
  }
  */

  public void writeWalls() {
    for (int i = 0; i < wall.length; i++) {
```

```
        System.out.println("(" + wall[i].getX1() + "," + wall[i].getY1() + ") to ("
                        + wall[i].getX2() + "," + wall[i].getY2() + ")");
      }
    }

    public String toString() {
      return "Building:  " + footprint.toString() + height;
    }
}
```

F. CLASS PLOS

```
/*
 * PLOS.java
 *
 * Created on October 18, 2003, 3:24 AM
 */

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.geom.*;
import simkit.*;
import simkit.stat.*;
import simkit.util.*;
import simkit.random.*;

public class PLOS {

  public static void main(String[] args) {

    // declare variables

    String inputString;
    FileReader inputFile;
    BufferedReader inputUnit;
```

```
StringTokenizer tokenizer;

int numberOfBuildings, numberOfCorners, range;
Building[] building;
Line2D[] wall;
int[] xPoints, yPoints;
double areaXLength, areaYLength, height;
SimpleStatsTally heightData = new SimpleStatsTally();
boolean inBuilding;
int i;

if (args.length == 0) {
  System.out.println("\nUsage:  java PLOS <filename>");
  return;
}

// read in the data

try {
  inputFile = new FileReader(args[0]);
  inputUnit = new BufferedReader(inputFile);
  inputString = inputUnit.readLine();
  tokenizer = new StringTokenizer(inputString," ");
  areaXLength = Integer.parseInt(tokenizer.nextToken());
```

```
        areaYLength = Integer.parseInt(tokenizer.nextToken());
        numberOfBuildings = Integer.parseInt(tokenizer.nextToken());

        building = new Building[numberOfBuildings];
        i = 0;
        while ((inputString = inputUnit.readLine()) != null) {
          tokenizer = new StringTokenizer(inputString," ");
          //System.out.println(i);
          numberOfCorners = Integer.parseInt(tokenizer.nextToken());
          wall = new Line2D[numberOfCorners];
          xPoints = new int[numberOfCorners];
          yPoints = new int[numberOfCorners];

          for(int j = 0; j < numberOfCorners; j++) {
            xPoints[j] = Integer.parseInt(tokenizer.nextToken());
            yPoints[j] = Integer.parseInt(tokenizer.nextToken());
            if (j > 0) {
              wall[j-1] = new Line2D.Double(xPoints[j-1], yPoints[j-1], xPoints[j],
yPoints[j]);
            }
          }

          wall[wall.length - 1] = new Line2D.Double(xPoints[wall.length - 1],
yPoints[wall.length - 1], xPoints[0], yPoints[0]);
          height = Double.parseDouble(tokenizer.nextToken());
```

```
        building[i] = new Building(new Polygon(xPoints, yPoints, numberOfCorners),
wall, height);
        heightData.newObservation(height);
        i++;
      }
      inputUnit.close();

    } catch(FileNotFoundException e) {
      System.out.println(e);
      return;

    } catch(IOException e) {
      System.out.println(e);
      return;
    }

    // print out some statistics on the building heights
    /*
    System.out.println("Max Building Height:  " + heightData.getMaxObs());
    System.out.println("Min Building Height:  " + heightData.getMinObs());
    System.out.println("Mean Building Height:  " + heightData.getMean());
    System.out.println("Standard Deviation:  " + heightData.getStandardDeviation());
    System.out.println();
    */
```

```
// run the simulation
//for(int a = 1; a <= 5; a++) {

Sensor sensor = new Sensor();
Target target = new Target();

RandomVariate u = RandomVariateFactory.getInstance(
  "Uniform", new Object[] {new Double(0.0), new Double(areaXLength)}, CongruentialSeeds.SEED[1]);
RandomVariate v = RandomVariateFactory.getInstance(
  "Uniform", new Object[] {new Double(0.0), new Double(areaYLength)}, CongruentialSeeds.SEED[5]);

int binSize;
binSize = 5;

SimpleStatsTally[] bin = new SimpleStatsTally[(int)(Math.ceil(Math.sqrt(areaXLength*areaXLength + areaYLength*areaYLength) / binSize) + 1)];

for(int k = 0; k < bin.length; k++) {
  bin[k] = new SimpleStatsTally();
}
```

```
for(int r = 0; r < 10000; r++) {
  do {
    inBuilding = false;
    i = 0;
    sensor.setLocation(new Point3D(u.generate(), v.generate(), 20.0));
    while (!inBuilding && i < building.length) {
      if (building[i].contains(sensor.getLocation())) {
        inBuilding = true;
      }
      i++;
    }
  } while (inBuilding);

  do {
    inBuilding = false;
    i = 0;
    target.setLocation(new Point3D(u.generate(), v.generate(), 0.0));
    while (!inBuilding && i < building.length) {
      if (building[i].contains(target.getLocation())) {
        inBuilding = true;
      }
      i++;
    }
  } while (inBuilding);
```

```
        range = (int)Math.floor(sensor.getLocation().XYDistanceFrom(target.getLocation())) / binSize;
        /*
        System.out.println(sensor.getLocation());
        System.out.println(target.getLocation());
        System.out.println(sensor.hasLOSTo(target, building));
        System.out.println(sensor.hasLineOfSightTo(target, building));
        */
        if (sensor.hasLOSTo(target, building)) {
          bin[range].newObservation(1.0);
        }
        else {
          bin[range].newObservation(0.0);
        }

    }


    for(int k = 0; k < bin.length; k++) {
      System.out.println(binSize*(k+1) + ", " + bin[k].getMean() + ", " + bin[k].getCount());
    }

    //}
```

```
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. CODE FOR GENERATING PLOS ESTIMATES WITH COMBAT XXI

```
/*
   File:       Location.java
 */

/* Copyright Notice ==========================================*
 * This file contains proprietary information of USATRAC-WSMR and USMC-MCCDC
 * Copying or reproduction without prior written approval is prohibited.
 * Copyright (c) 1999 - 2004 ============*/

// PACKAGE
package cxxi.util.geom;

// IMPORTS
import cxxi.util.geom.*;
import cxxi.util.tools.Units;
import cxxi.environmentservices.*;
import simkit.*;
import simkit.stat.*;
import simkit.random.*;

import cxxi.coremodel.communications.msg.Sharable;

import java.awt.geom.Point2D;
import java.util.*;

import net.onesaf.services.data.dm.objects.ecs.coordinate.*;

/*
 TODO  When new env services are hooked up add code to initialize elevation to use the new terrain services.
 There should be a static method that does this initialization
   * as soon as the env services have been initialized.
*/

/** This class provide a way of specifying locations in the battle space.  The representation
 * used is based on the geocentric coordinate system.  For this reason x, y and z cannot be
 * interpreted to mean map offsets and elevation.  Methods are provided to get this type
 * of information based on a reference ellipsoid.
```

```
         *
         *
         * History:  This class is based on the LocationXYZ class developed by Michael
Shattuck in
         * Aug 1999.  Some of the methods used here are lifted from that class.
         *
         **/
        public class Location implements Cloneable, Sharable

        {
           /*
************************************************************************
*
            *
            * Static attribute for the class
            *
            *
************************************************************************
*/
           /**
            * Determine if the validation of the value in a locatio is done - validation
            * makes sure that the value of a location is set before use.  If this flag is
            * false, new locations will have default values of 0.0 otherwise default values
            * will be Double.NaN.  This attribute can be used by other classes in the
            * geom package.
            */
           static final boolean validationOK = false;

           /** flag to control if object pooling should be used - this should only
            * be used during development of this mechanism - once it is has been fully
            * tested the pooling should be left on
            */
           private static boolean ObjectPoolingDisabled = true;

           /** The structure to hold the pool of locations to allow for the recycling of
            * location objects.  Care should be taken when recycling locations since the
            * information in the object is lost.  Recycling should be mainly used for temp
            * locations that are not passed to other parts of the code.
            */
           private static Vector locationPool = new Vector();

           /** This is a handle to the environment services that are being used.
            */
           private static cxxi.environmentservices.ModTerrainService mts = null;

           /** This attribute holds the grid zone to use when covering to Lat Long - this
```

```
     * is here only until the internal conversion to GCC is done - at that time
     * this will no longer be needed.
     *
     * !!!!!!! Note !!!!!! for now it is assumed that the entire playbox is in a
     * single grid zone. If this assumtion is violated Locations will not be
correct!!!!
     *
     */
    private static byte gridZone = -100;

    // attribute to allow having dummy locations outside a valid play zone
    private static double dummyLong = Double.NaN;

    /**
    * Default Constructor
    * All coordinates = 0.0.
    **/
    public
    Location()
    {
       super();

       if (validationOK)
       {
          this.setX(Double.NaN);
          this.setY(Double.NaN);
          this.setZ(Double.NaN);
       }
       else
       {
          this.setX(0.0);
          this.setY(0.0);
          this.setZ(0.0);
       }

    }

    /** Return a location set to the latitude and longitude passed in. The elevation
     * is clamped to the surface of the terrain skin.
     *
     * @param aLatitude the latitude to use
     * @param aLongitude the longitude to use
     *
     * @return a location object set to the values passed in
     */
```

```
        public static Location getInstanceInLatLon(double aLatitude, double
aLongitude)
        {
           Location temp = getALocation();
           temp.setLatitudeAndLongitude(aLatitude, aLongitude);

           return temp;
        }

        /** Return a location set to the latitude and longitude passed in.  The elevation
         * is set to the value passed in.
         *
         * @param aLatitude the latitude to use
         * @param aLongitude the longitude to use
         * @param elevation the elevation above mean sea level
         *
         * @return a location object set to the values passed in
         */
        public static Location getInstanceInLatLon(double aLatitude, double
aLongitude, double elevation)
        {
           Location temp = getALocation();
           temp.setLatitudeAndLongitude(aLatitude, aLongitude, elevation);

           return temp;
        }

        /** Return a location initialized to the values passed in.  The location returned
may
         * either be new or taken from the pool.
         */
        public static Location getInstance(double xValue, double yValue, double
zValue)
        {
           Location temp = getALocation();
           temp.setX(xValue);
           temp.setY(yValue);
           temp.setZ(zValue);

           return temp;
        }

        /** Return a location initialized to the value passed in.  The location returned
may
         * either be new or taken from the pool.
         */
```

```
    public static Location getInstance(Location location)
    {
       Location temp = getALocation();
       temp.setLocation(location);
       return temp;
    }

    /** This method returns a "new" location - this may either be newly created or
it
     * can come from the object pool.  If there are objects in the pool, these are
used
     * otherwise a new object is created.
     */
    private static Location getALocation()
    {
       if (locationPool.isEmpty())
          return new Location();
       else
       {
          Location temp = (Location)locationPool.remove(0);
          temp.isInPool = false;

          return temp;
       }
    }

    /** return a location that is in the current gridzone but outside of any
reasonable
     * playbox.
     */
    public static Location getDummyLocation()
    {
       return getInstanceInLatLon(-85.0, dummyLong, 0.0);
    }

    /** This method returns this instance of Location to the pool of Location
objects
     * that can be reused.  This method should be used only when it is certain that
this
     * location is not used by any other object.  The location values of this object
are
     * erased.  After this method is called, the reference to this object should be set
     * to null or overwritten. For Example:
     * <br>
     * <CODE>
     *  aLocation.returnToPool();
```

```
     *  alocation = null;
     * </CODE>
     */
    public void returnToPool()
    {
        if (ObjectPoolingDisabled) return;

        if (this.isInPool)
        {
            System.err.println("/n****WARNING: Location Pool" +
            " - trying to return a location that is already in the pool " +
            Integer.toHexString(this.hashCode()) + "/n/n");
        }
        else
        {
            this.setX(Double.NaN);
            this.setY(Double.NaN);
            this.setZ(Double.NaN);
            this.locked = false;
            this.isInPool = true;
            this.latLonIsSet = false;
            this.isClampedToTerrainSkin = false;

            locationPool.add(this);
        }
    }

    /**
     * Determin if this location has valid values.
     */
    public boolean isIllDefined()
    {
        if (validationOK)
            return (Double.isNaN(myX) || Double.isNaN(myY) ||
Double.isNaN(myZ));
        else
            return false;
    }

    /**
     */
    public double getLatitude()
    {
        if (!latLonIsSet)
        {
```

```
            if (mts == null)
               System.err.println("\n *** ERROR cannot get at terrain services in
Location\n");
            else
            {
               geotransform.coords.Utm_Coord_3d utm3d = new
geotransform.coords.Utm_Coord_3d();
               utm3d.x = myX;
               utm3d.y = myY;
               utm3d.z = myZ;
               utm3d.hemisphere_north = true;
               utm3d.zone = gridZone;

               geotransform.coords.Gdc_Coord_3d gdc3d = new
geotransform.coords.Gdc_Coord_3d();
               geotransform.transforms.Utm_To_Gdc_Converter.Convert(utm3d,
gdc3d);
               this.latitude = gdc3d.latitude;
               this.longitude = gdc3d.longitude;
               this.elevation = gdc3d.elevation;
               latLonIsSet = true;
            }
         }
         return this.latitude;
      }

      /**
       */
      public double getLongitude()
      {
         getLatitude();
         return this.longitude;
      }

      /** This returns the distance above sea level in meters - this should
       * be the same value that is returned by getAltitudeMSL.  The reason for two
       * methods is that refering to elevation make more sence when dealing with
       * things on the ground and using altitude is more meaningfull for aircraft.
       */
      public double getElevation()
      {
         return myZ;
      }

      /**
       */
```

```
public double getNorthing()
{
   return myY;
}

/**
 */
public double getEasting()
{
   return myX;
}

/**
 */
public int getGridZone()
{
   return myGridZone;

}

public boolean isNorthernHemisphere()
{
   return northernHemisphere;
}

/** return if this location has been clamped to the surface of the earth
 */
public boolean isClampedToTerrainSkin()
{
   return isClampedToTerrainSkin;
}



/**
 */
public double getAltitudeMSL()
{
   return getElevation();
}

/** Modify this location so that its projection onto the surface of the Earth
 * does not change while its distance above mean sea level is the distance
passed in
 *
 * @param altitude the new altitude value in meters
```

```
     */
    public void setAltitudeMSL(double altitude)
    {
       //****** Implementation NOTE!!!!!
       //  latLonIsSet does not need to be changed because this transformation does
not
       // affect the lat and lon values

       if(this.isIllDefined())
          throw new IllegalStateException("Attempted to use ill-defined Location");
       // Cannot modify a locked object
       if(this.isLocked()) {
          throw new IllegalStateException("Attempted modify locked Location
object.");
       }
       myZ = altitude;
       elevation = myZ;
       isClampedToTerrainSkin = false;
    }

    /** This method sets the "elevation" of this point to be height meters above
     * the terrain skin.  This method is intended to be used when locations need to
     * be specified that are above the terrain skin by a small amount such as sensor
     * locations or positions inside of multistoried buildings.
     *
     * @param height the translated point should be above the surface (meters)
     */
    public void setHeightAboveTerrainSkin(double height)
    {
       //****** Implementation NOTE!!!!!
       //  latLonIsSet does not need to be changed because this transformation does
not
       // affect the lat and lon values
       clampToTerrainSkin();
       myZ += height;  // this needs to be redone when the GCC based location is
integrated.
       elevation = myZ;
       isClampedToTerrainSkin = false;
    }

   /**
    * Clamp the elevation value of this location to the elevation at the surface
    * of the earth.  This method goes to the current environment services and gets
the
    * elevation of the closest point on the surface of the earth to this point and
    * set the elevation to it.
```

```
        *
        */
        public void clampToTerrainSkin()
        {
           //****** Implementation NOTE!!!!!
           // latLonIsSet does not need to be changed because this transformation does
not
           // affect the lat and lon values

           if(this.isIllDefined())
              throw new IllegalStateException("Attempted to use ill-defined Location");

           // Cannot modify a locked object
           if(this.isLocked()) {
              throw new IllegalStateException("Attempted modify locked Location
object.");
           }

           // make sure we have initialized the environmet services
           if (mts == null)
           {
              mts                                                                    =
cxxi.environmentservices.EnvironmentServices.ModTerrainService();
              if (mts == null)
              {
                 throw new IllegalStateException(" **** Error in Location: " +
                 "Tried to use clampToTerrainSkin before terrain services were
initialized\n" +
                 "         This Location elevation not set to the terrain skin value\n\n");
              }
           }
           cxxi.environmentservices.Elevation localElevation =
                    mts.getElevation(this);
           myZ = localElevation.getTerrainHeight();
           elevation = myZ;
           isClampedToTerrainSkin = true;
        }


        /**
        * Determine if two XYZ coordinates are equal.
        * @param x1 x-coordinate of first position.
        * @param y1 y-coordinate of first position.
        * @param z1 z-coordinate of first position.
        * @param x2 x-coordinate of first position.
        * @param y2 y-coordinate of first position.
```

```
* @param z2 z-coordinate of first position.
* @return true if coordinates are equal
**/
public static boolean
CoordinatesEqual3D(double x1,
                   double y1,
                   double z1,
                   double x2,
                   double y2,
                   double z2)
{
   boolean equal = false;

   if (StrictMath.abs(x1 - x2) <= Double.MIN_VALUE &&
       StrictMath.abs(y1 - y2) <= Double.MIN_VALUE &&
       StrictMath.abs(z1 - z2) <= Double.MIN_VALUE)
   {
      equal = true;
   }

   return equal;
}

/**
* Get the x coordinate.
* <p>
* @return the x-coordinate of this location.
**/
double
getX()
{
   return myX;
}

/**
* Get the y coordinate.
* <p>
* @return the y-coordinate of this location.
**/
double
getY()
{
   return myY;
}

/**
```

```
     * Get the z coordinate.
     * <p>
     * @return the z-coordinate of this location.
     **/
     double
     getZ()
     {
        return myZ;
     }

    /**
     * Returns the distance from this Location to a specified Location.
     * This is the same as the Euclidean 3D distance.
     * <p>
     * Distance is the degree or amount of separation between two points,
     * lines, surfaces, or objects; or an extent of an area or an advance
     * along a route measured linearly; or an extent of space measured other
     * than linearly; or an extent of advance from a beginning.
     * <p>
     * @param toLoc The other location.
     * @return instance of Distance that represents the distance.
     **/
     public Distance
     distanceTo(Location toLoc)
     {

        double dist;

        if(this.isIllDefined())
           throw new IllegalStateException("Attempted to use ill-defined Location");
        if(toLoc.isIllDefined())
           throw new IllegalStateException("Attempted to use ill-defined Location as
an input argument");
        double distSq;
        Location that = (Location) toLoc;

        double deltaX = this.getX() - that.getX();
        double deltaY = this.getY() - that.getY();
        double deltaZ = this.getZ() - that.getZ();

        dist = StrictMath.sqrt((deltaX * deltaX) +
                               (deltaY * deltaY) +
                               (deltaZ * deltaZ));

        return Distance.getInstance(dist);
     }
```

```
        public double
         groundDistanceTo(Location toLoc)
         {

            if(this.isIllDefined())
               throw new IllegalStateException("Attempted to use ill-defined Location");
            if(toLoc.isIllDefined())
               throw new IllegalStateException("Attempted to use ill-defined Location as
an input argument");

            Location that = (Location) toLoc;

            double deltaX = this.getX() - that.getX();
            double deltaY = this.getY() - that.getY();


            return StrictMath.sqrt((deltaX * deltaX) +
                              (deltaY * deltaY));

         }

        /**
         * Returns the square of the distance from this Location to another.
         * This is the same as the square of the Euclidean 3D distance.
         * <p>
         * Distance is the degree or amount of separation between two points,
         * lines, surfaces, or objects; or an extent of an area or an advance
         * along a route measured linearly; or an extent of space measured other
         * than linearly; or an extent of advance from a beginning.
         * <p>
         * @param toLocation The other location.
         * @return instance of Distance that represents the distance.
         **/
         public Distance
         distanceToSqrd(Location toLoc)
         {
            if(this.isIllDefined())
               throw new IllegalStateException("Attempted to use ill-defined Location");
            if(toLoc.isIllDefined())
               throw new IllegalStateException("Attempted to use ill-defined Location as
an input argument");
            double distSq;

            Location that = (Location) toLoc;
```

```
            double deltaX = this.getX() - that.getX();
            double deltaY = this.getY() - that.getY();
            double deltaZ = this.getZ() - that.getZ();

            distSq = (deltaX * deltaX) +
                    (deltaY * deltaY) +
                    (deltaZ * deltaZ);

            return Distance.getInstance(distSq);
        }


        /**
         * Calculate the direction from this location to another.
         * <p>
         * Direction is the line or course on which something is moving or is
         * aimed to move or along which something is pointing or facing.
         * <p>
         * @param toLoc The other location.
         * @return The direction from this location to toLoc.
         **/
        public Direction
        directionTo(Location toLoc)
        {
            if(this.isIllDefined())
                throw new IllegalStateException("Attempted to use ill-defined Location");
            if(toLoc.isIllDefined())
                throw new IllegalStateException("Attempted to use ill-defined Location as
an input argument");

            double distSq;
            double deltaX;
            double deltaY;
            double deltaZ;

            Location that = (Location) toLoc;

            deltaX = that.getX() - this.getX();
            deltaY = that.getY() - this.getY();
            deltaZ = that.getZ() - this.getZ();

            Direction direct = new Direction(deltaX, deltaY, deltaZ);

            return direct;
        }
```

```
        /**
        * Calculate a new location based on the direction and distance from
        * this location.
        * <p>
        * @param direction The direction to the new location.
        * @param distance The distance to the new location.
        * @return The new location.
        **/
        public Location
        getTranslatedLocation(Direction direction,
                              Distance distance)
        {
           if(this.isIllDefined())
              throw new IllegalStateException("Attempted to use ill-defined Location");
           if(direction.isIllDefined())
              throw new IllegalStateException("Attempted to use ill-defined Direction
as an argument");
           if(distance.isIllDefined())
              throw new IllegalStateException("Attempted to use ill-defined Distance as
an argument");

           double distSq;
           double[] dirCosines =
              direction.getDirectionCosines();

           double xOffset =
              distance.getMeters() * dirCosines[0];

           double yOffset =
              distance.getMeters() * dirCosines[1];

           double zOffset =
              distance.getMeters() * dirCosines[2];

           return getInstance(this.getX() + xOffset,
                              this.getY() + yOffset,
                              this.getZ() + zOffset);
        }


        /**
        * Calculate a new location based on the direction and distance from
        * this location.
        * <p>
        * @param offset from this location to the new location.
        * @return The new location.
```

```
**/
public Location
getTranslatedLocation(Offset anOffset)
{
  return this.getTranslatedLocation(
                    anOffset.getDirection(), anOffset.getDistance());
}
/**
 * Calculate a new location offest from this location using a velocity and
 * time.  This assumes that the velocity is constant for the full time period.
 *
 * @param aVelocity the velocity to use
 * @param sTime the time in seconds that the movement is to take place
 *
 * @return the new location
 */
public Location
getTranslatedLocation(Velocity aVelocity,
                      double aTime)
{

   if(this.isIllDefined())
      throw new IllegalStateException("Attempted to use ill-defined Location");
   if(aVelocity.isIllDefined())
      throw new IllegalStateException("Attempted to use ill-defined Velocity as
an argument");
   if(aTime < 0.0)
      throw new IllegalStateException("Attempted to use negative time as an
argument");

   Direction aDir = aVelocity.getDirection();
   Distance aDist = Distance.getInstance(aVelocity.getSpeed() * aTime);
   return getTranslatedLocation(aDir, aDist);
}

/**
* Assign a new location to this instance.
* <p>
* @param location The new location.
**/
public void
setLocation(Location location)
{
   // Cannot modify a locked object
   if(this.isLocked())
   {
```

```
            throw new IllegalStateException("Attempted modify locked Location
object.");
        }

        if (location != null)
        {
            this.myX = location.getX();
            this.myY = location.getY();
            this.myZ = location.getZ();
            this.latLonIsSet = location.latLonIsSet;
            this.elevation = location.elevation;
            this.latitude = location.latitude;
            this.longitude = location.longitude;
            this.isClampedToTerrainSkin = location.isClampedToTerrainSkin;
        }
    }

    /**
    * Set a new value for the x-coordinate.
    * <p>
    * @param xValue : the x-coordinate to set.
    **/
    void
    setX(double xValue)
    {
        // Cannot modify a locked object
        if(this.isLocked()) {
            throw new IllegalStateException("Attempted modify locked Location
object.");
        }

        myX = xValue;
        latLonIsSet = false;
        isClampedToTerrainSkin = false;
    }

    /**
    * Set a new value for the y-coordinate.
    * <p>
    * @param yValue : the y-coordinate to set.
    **/
    void
    setY(double yValue)
    {
        // Cannot modify a locked object
        if(this.isLocked()) {
```

```
            throw new IllegalStateException("Attempted modify locked Location
object.");
        }

        myY = yValue;
        latLonIsSet = false;
        isClampedToTerrainSkin = false;
    }

    /** Set a new value for the z-coordinate.
     * <p>
     * @param ZValue the z value
     */
    void
    setZ(double ZValue)
    {
        // Cannot modify a locked object
        if(this.isLocked()) {
            throw new IllegalStateException("Attempted modify locked Location
object.");
        }

        myZ = ZValue;
        latLonIsSet = false;
        isClampedToTerrainSkin = false;
    }

    /**
     * Set this location to the lat lon passed in and clamp the elevation to
     * the terrain skin.
     */
    private void setLatitudeAndLongitude(double aLatitude, double aLongitude)
    {
        setLatitudeAndLongitude( aLatitude, aLongitude, 0.0);
        this.clampToTerrainSkin();
    }

    /**
     * Set this location to the lat lon passed in and set the elevation to
     * the value pased in.
     */
    private void setLatitudeAndLongitude(double aLatitude, double aLongitude,
double anElevation)
    {
        geotransform.coords.Gdc_Coord_3d gdc3d =
```

```
                new geotransform.coords.Gdc_Coord_3d(aLatitude, aLongitude,
elevation);

            geotransform.coords.Utm_Coord_3d utm3d = new
geotransform.coords.Utm_Coord_3d();

            geotransform.transforms.Gdc_To_Utm_Converter.Convert(gdc3d, utm3d);

            this.setX(utm3d.x);
            this.setY(utm3d.y);
            this.setZ(anElevation);
            this.myGridZone = utm3d.zone;
            this.northernHemisphere = utm3d.hemisphere_north;

            if (gridZone < -60)
               gridZone = utm3d.zone;
            else
               if (gridZone != utm3d.zone)
               {
                  System.err.println("\n**** ERROR with locations - playbox appears to
span grid zones!!! "
                  + gridZone + " and  " + utm3d.zone);
               }

            if (Double.isNaN(dummyLong))
               dummyLong = aLongitude;

            latitude = aLatitude;
            longitude = aLongitude;
            elevation = anElevation;
            latLonIsSet = true;
         }

         /**
         * Check if this location is the same as another.
         * @param object should be an instance of Location.
         * @return true if the passed in object is a location
         * with the same coordinates.
         **/
         public boolean
         equals(Object object)
         {
            boolean locationsEqual = false;

            if (object instanceof Location)
            {
```

```
            Location thatLocation =
               (Location) ((Location) object);

            if (CoordinatesEqual3D(getX(),
                                   getY(),
                                   getZ(),
                                   thatLocation.getX(),
                                   thatLocation.getY(),
                                   thatLocation.getZ())
             )
              {
                 locationsEqual = true;
              }
          }

          return locationsEqual;
        }


        /* HashCode is implemented because equals is overwritten in this class.
         * This implementation based on "Effective Java" --Jason Bloch, Sun 2001, Ch
3, pp 38-39
         * The hashcode uses Location hashcode and the route nodetype hashcode.
         *
         * @return int hashCode that is unique to this object and differs in the near
fields.
         */
        public int hashCode()
        {
           if(myHashCode == 0)
           {
              long xCode = Double.doubleToLongBits(myX);
              long yCode = Double.doubleToLongBits(myY);
              long zCode = Double.doubleToLongBits(myZ);
              int result = 17;
              result = 37*result + (int)(xCode^(xCode>>>32));
              result = 37*result + (int)(yCode^(yCode>>>32));
              result = 37*result + (int)(zCode^(zCode>>>32));
              myHashCode = result;
           }
           return myHashCode;
        }


        void
        clear()
```

```
        {
            // Cannot modify a locked object
            if(this.isLocked()) {
                throw new IllegalStateException("Attempted modify locked Location
object.");
            }

            myX = myY = myZ = Double.NaN;
            latLonIsSet = false;
            isClampedToTerrainSkin = false;
        }


        /**
        * return a string representation of this location.
        * Overrides toString method from Object.
        **/
        public String
        toString()
        {
            java.text.DecimalFormat df = new java.text.DecimalFormat("0.00");
            StringBuffer stringBuf = new StringBuffer("Location(");
            stringBuf.append(df.format(this.getX()) + ", ");
            stringBuf.append(df.format(this.getY()) + ", ");
            stringBuf.append(df.format(this.getZ()) + ")");
            return new String(stringBuf);
        }


        /*
================================================== *
          SHAREABLE METHODS
         *
================================================== */

        /**
         * This method is required to implement Sharable.  It locks the object
         * preventing further modification of it's data.
         *
         * @see cxxi.coremodel.communications.Sharable#lock
         */
        public void lock() {
            locked = true;
        }

        /**
```

```
     * This method is required to implement Sharable.  It returns
     * the lock state of the object.
     *
     * @see cxxi.coremodel.communications.Sharable#isLocked
     * @return lock state of the object
     */
    public boolean isLocked()  {
       return locked;
    }

    /**
     * This method is required to implement Sharable.  Returns
     * a deep copy of the object with locked set to false.
     *
     * @see cxxi.coremodel.communications.Sharable#getMutableCopy
     * @return a deep copy of the locked object
     */
    public Object getMutableCopy() {
       Location newLocation = null;
       try {
          newLocation = (Location)this.clone();
          newLocation.locked = false;
       }
       catch(Exception e)  {
          // ignore as this should never occur
       }
       return newLocation;
    }

    /*
================================================================ *
      CLONEABLE METHODS
     *
================================================================ */

    public Object clone() throws CloneNotSupportedException {
       Location newLocation = (Location)super.clone();

       return newLocation;
    }

    /*
================================================================ *
     *  Static support methods
     *
     *  Implementation Note:  perhaps these methods should be in some
```

```
     *  kind of support class.  If there are only a few this is
     *  probably a good place for them.  But if the number grows
     *  they should probably me in their own class.
     *
     *
==========================================================  */

    /**
    * Determine if two XY coordinates are equal.
    * @param x1 x-coordinate of first position.
    * @param y1 y-coordinate of first position.
    * @param x2 x-coordinate of first position.
    * @param y2 y-coordinate of first position.
    * @return true if coordinates are equal
    **/
    public static boolean
    CoordinatesEqual2D(double x1,
                       double y1,
                       double x2,
                       double y2)
    {
       boolean equal = false;

       if (StrictMath.abs(x1 - x2) <= Double.MIN_VALUE &&
           StrictMath.abs(y1 - y2) <= Double.MIN_VALUE)
       {
          equal = true;
       }

       return equal;
    }


    /** Compute the mathematical center of mass of the locations passed in.  No
     * assumptions are made about the meaning of x, y and z (i.e. z is NOT
assumed
     * to be elevation).
     *
     * @param locations the list of locations whose center of mass should be
     * found - all elements of the list should be of Location type.
     * @return a point that is the mathematical center of the points. A null is
     * returned if the list passed in is empty.
     */
    public static Location calculateCenterOfMass(List locations)
    {
       if (locations.size() < 1)
```

```
            return null;

          // establish a center point
           double xctr = 0;
           double yctr = 0;
           double zctr = 0;
           Location tempLoc = null;

           // iterate through the list summing the coordinates
           for(int i=0;i<locations.size();i++)
           {
              tempLoc = (Location)locations.get(i);

              xctr = xctr + tempLoc.getX();
              yctr = yctr + tempLoc.getY();
              zctr = zctr + tempLoc.getZ();
           }// end i for

           // divide by the number of points to get arithmatic center
           xctr = xctr/((double)locations.size());
           yctr = yctr/((double)locations.size());
           zctr = zctr/((double)locations.size());

           Location centerOfMass = Location.getInstance(xctr, yctr, zctr);

           return centerOfMass;

        }// end calculateCenterOfMassMethod method

        // Static block to initialize the converstions needed for this class
        static
        {
           // all conversion intitializations should go here
           geotransform.transforms.Gdc_To_Utm_Converter.Init();
           geotransform.transforms.Utm_To_Gdc_Converter.Init();
        }

        /*
========================================================== *
           ATTRIBUTES
         *
========================================================== */

        // for the sharable interface
        private boolean locked = false;
```

```
protected double myX = 0.0;
protected double myY = 0.0;
protected double myZ = 0.0;

private byte myGridZone = -1;
private boolean northernHemisphere = true;

// these attributes are used to cut down on the transformations needed
// when getting the value of this location in terms of other coordinate systems
private double latitude = Double.NaN;
private double longitude = Double.NaN;
private double elevation = Double.NaN;

// this flag indicates if the lat lon values have been set to match the current
// x, y, z.  Whenever x, y, or z are changed this flag should be set to false.
When
// a call is made to get a lat or lon value the conversion is done and the result is
// saved and this flage is set to true so that if the values are needed again the
// convrsion does not need to be done again.
private boolean latLonIsSet = false;

private int myHashCode;

/** flag to indicate if this location is already in the pool
 */
private boolean isInPool = false;

// Flag to indicate if this location has been forced to be on the terrain skin
private boolean isClampedToTerrainSkin = false;

      /**
* class test.
*/
public static void main(String[] args)
{
   Location temp, temp2;


   // initiialize the terrain for testing
   StringBuffer errors = new StringBuffer();
   cxxi.environmentservices.EnvironmentServices.Initialize(
      "C:\\Program
Files\\CXXI\\cxxibasedata\\envir\\CommercialRibbonSmoothCropped.ele",
      "dummy", errors);
   System.out.println(errors);
   mts = cxxi.environmentservices.EnvironmentServices.ModTerrainService();
```

```
            if (mts != null)
            {
               // set up box to sample
               /*
                *  SWLatitude = 27.0476678
                   SWLongitude = 57.330884
                   NELatitude = 27.0544301
                   NELongitude = 57.338477
               */

               double llLat = 27.0476678;
               double llLong = 57.330884;
               Location upperRight = getInstanceInLatLon(27.0544301, 57.338477);
               Location lowerLeft = getInstanceInLatLon(llLat, llLong);
               double latRange = upperRight.getLatitude() - lowerLeft.getLatitude();
               double longRange = upperRight.getLongitude() -
lowerLeft.getLongitude();

               SimpleStatsTally[] bin = new SimpleStatsTally[250];

               int binSize;
               binSize = 5;

               int range;

               for(int i = 0; i < bin.length; i++) {
                  bin[i] = new SimpleStatsTally();
               }

               Location sensor = new Location();
               Location target = new Location();

               RandomVariate u = RandomVariateFactory.getInstance("Uniform", new
Object[] {new Double(0.0), new Double(1.0)}, CongruentialSeeds.SEED[5]);
               RandomVariate v = RandomVariateFactory.getInstance("Uniform", new
Object[] {new Double(0.0), new Double(1.0)}, CongruentialSeeds.SEED[7]);


               /*      System.out.println("UR, LL, Viewer: "+
                     upperRight.getLatitude() + ", " + upperRight.getLongitude()  + ", " +
                     lowerLeft.getLatitude() + ", " + lowerLeft.getLongitude()  + ", " +
                     viewer.getLatitude() + ", " + viewer.getLongitude());
                     System.out.println("Viewer Altitude: "+viewer.getAltitudeMSL());
                */
               for (int i = 0; i < 1000000; i++)
               {
```

```
                    sensor = getInstanceInLatLon(llLat + (latRange * v.generate() ), llLong
+ (longRange * u.generate() ));
                    sensor.setZ(780.0);

                    do {
                      target = getInstanceInLatLon(llLat + (latRange * v.generate() ),
llLong + (longRange * u.generate() ));
                    } while (target.getAltitudeMSL() > 280.0);

                    //System.out.print("Sensor Altitude - " + sensor.getZ());
                    //System.out.print("Target Altitude - "+ target.getAltitudeMSL());
                    //System.out.println("Distance - " + sensor.groundDistanceTo(target));
                    /*if (mts.isLineOfSight(sensor, target))
                       System.out.println(" TRUE - "+ target.getLatitude() + ", " +
target.getLongitude() +", "+
                       lowerLeft.getLatitude() + ", " + lowerLeft.getLongitude());
                    else
                       System.out.println(" FALSE - "+ upperRight.getLatitude() + ", " +
upperRight.getLongitude()
                          + ", " + target.getLatitude() + ", " + target.getLongitude());
                     */

                    range = (int)Math.floor(sensor.groundDistanceTo(target)) / binSize;

                    if (mts.isLineOfSight(sensor, target)) {
                       bin[range].newObservation(1.0);
                    }
                    else {
                       bin[range].newObservation(0.0);
                    }
                 }

                 for(int i = 0; i < bin.length; i++) {
                    System.out.println(binSize*(i+1) + ", " + bin[i].getMean() + ", " +
bin[i].getCount());
                 }
              }
              else
              {
                 System.err.println("could not init terrain file");
              }


           }

        } // Location class.
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Army Modeling and Simulation Resource Repository [Army MSRR]. Army Modeling and Simulation Office. 02 June 2004 http://www.msrr.army.mil/index.cfm.

"Artillery Effects In A MOUT Environment." Fordyce, David. Presentation given at Military Operations Research Society Symposium, June, 2003.

Champion, Danny C., et al. *The Prediction of Line of Sight (LOS) in Vegetation Using Remote Sources*. White Sands Missile Range: TRADOC Analysis Center – White Sands Missile Range, 2002.

Champion, Danny C., Kent G. Pankratz, and Louis A. Fatale. *The Effects of Different Line-of-Sight Algorithms and Terrain Elevation Representations on Combat Simulations*. White Sands Missile Range: TRADOC Analysis Center – White Sands Missile Range, 1995.

"Department of Defense Dictionary of Military and Associated Terms." Department of Defense Joint Publication 1-02 [JP 1-02]. 12 April 2001 (as amended through 23 March 2004).

Driels, Morris, and Bard Mansager. *Detection of Multiple Targets By Multiple Observers*. Monterey: Naval Postgraduate School, 2001.

Ellefsen, Richard. *Urban Terrain Zone Characteristics*. Aberdeen Proving Ground: United States Army Human Engineering Laboratory, 1987.

Gerald, Curtis F. and Patrick O. Wheatley. *Applied Numerical Analysis*. Sixth ed. San Luis Obispo: Addison-Wesley, 1999.

Notes on S-Plus: A Programming Environment for Data Analysis and Graphics [Notes on S-Plus]. The University of Adelaide. 02 June 2004 http://www.isds.duke.edu/computing/S/Snotes/Splus.html.

Overview (Java 2 Platform SE v1.4.2). Sun Microsystems, Inc. 01 May 2004 http://java.sun.com/j2se/1.4.2/docs/api/index.html.

"Probability of Line-of-Sight (PLOS) and Roughness Enhancements for JWARS 1.5." Blacksten, H. Ric, Charles D. Burdick, and Mihaly G. Grell. Presentation given at INFORMS/MAS, November, 2002.

"Probability of Line-of-Sight (PLOS) and Roughness for JWARS 1.5." Blacksten, H. Ric, and Charles D. Burdick. Presentation given at Line-of-Sight Technical Working Group, January, 2004.

"Rapid Generation of Synthetic Urban Environments." Gray, Connie B. and McKeown, David M. Presentation given at MOUT FACT Project Implementation Meeting, May, 2002.

Ross, Sheldon M. *Probability Models*. Eighth ed. San Diego: Academic Press, 2003.

Simkit Home Page. Buss, Arnold, Naval Postgraduate School. 01 May 2004 http://diana.gl.nps.navy.mil/Simkit/.

Stone, George F. and Gregory A. McIntyre. *The Joint Warfare System (JWARS): A Modeling and Analysis Tool for the Defense Department*. Arlington: Joint Warfare System Office, 2001.

Taylor, James G. *Force on Force Attrition Modeling*. *Lanchester Models of Warfare*. 2 Vols. Alexandria: Military Applications Section of the Operations Research Society of America, 1983.

Tutton, Stephanie J. *Optimizing the Allocation of Sensor Assets for the Unit of Action*. MS Thesis. Monterey: Naval Postgraduate School, 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendelton, California

7. Director, Studies and Analysis Division, MCCDC, Code C45
   Quantico, Virginia

8. Thomas M. Cioppa
   TRADOC Analysis Center – Monterey
   Monterey, California

9. Craig Rasmussen
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California

10. Donovan Phillips
    TRADOC Analysis Center – Monterey
    Monterey, California

11. Thomas Lucas
    Department of Operations Research
    Naval Postgraduate School
    Monterey, California

12. Arnold Buss
MOVES Department
Naval Postgraduate School
Monterey, California

13. Mary Jo Mlakar
5276 Marian Drive
Lyndhurst, OH 44124

14. Jennifer Mlakar
102 Aston Court
Stafford, VA 22554